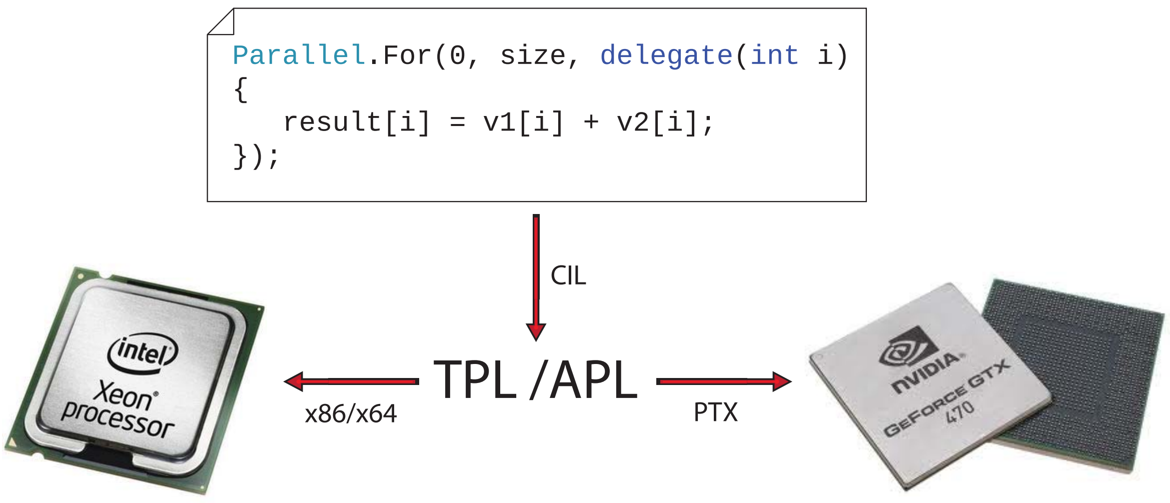
Accelerated Parallel Library

A .NET library for GPGPU programming
using the Task Parallel Library abstraction



```
Parallel.For(0, size, delegate(int i)
{
    result[i] = v1[i] + v2[i];
});
```

Søren Alsbjerg Hørup
Søren Andreas Juul
Henrik Holtegaard Larsen

June 7th, 2011

Software Engineering
Master Thesis
Aalborg University

Aalborg University
Department of Computer Science
Selma Lagerlöfs Vej 300
9220 Aalborg
Telephone:(45)99409940
http://www.cs.aau.dk

Title:
Accelerated Parallel Library

Project period:
February 2nd, 2011 to
June 7th, 2011

Theme:
GPGPU Programming

Group:
f11d611a
Software Engineering
Database and Programming Technologies

Authors:
Søren Alsbjerg Hørup
Søren Andreas Juul
Henrik Holtegaard Larsen

Supervisor:
Lone Leth Thomsen

Abstract:

In this report, we document the development of the Accelerated Parallel Library (APL), a library for General-Purpose computations on Graphics Processing Units (GPGPU) in languages which make use of the Common Language Infrastructure (CLI), such as C# or VB.NET.
APL aims to expose the same programming interface as the parallel loops of the `Parallel` class which is part of the Task Parallel Library (TPL) in the .NET 4.0 framework. This allows existing code, which uses TPL, to take advantage of a Graphics Processing Unit (GPU) with very few code changes.
To execute code on the GPU, APL makes use of reflection to access the Common Intermediate Language (CIL) code of an application which is then Just-in-Time (JIT) compiled to equivalent Parallel Thread Execution (PTX) code and invoked on the GPU using the Compute Unified Device Architecture (CUDA) Driver API.
At present, the library supports the CIL opcodes required to run our benchmark suite containing four benchmarks. The benchmarks show that APL is in general slower than the corresponding implementation in handwritten CUDA C, but in the case of the Vector Addition and the Black Scholes benchmarks, APL is shown to give a speedup of 1.03x and 1.02x when compared to the CUDA C implementation in steady state. The benchmarks also shows that APL in most cases outperforms TPL and in one case gives a speedup of 82x.

Copies: 6

Total pages: 143

Appendices: 6

Project finished: 7th of June 2011

1 Preface

Style Guide

The following style is used throughout this report:

Citations are represented as a pair of angle-brackets containing a number. The number refers to a number in our bibliography. Used as *this section is based on [1]* means that the entire section is based on the mentioned source, unless other sources are explicitly stated.

A citation at the end of a sentence, but right before the full stop, means that the citation is used exactly for that sentence. A citation after a full stop means that the citation is applied to the whole paragraph, i.e. more than one sentence.

Citing a specific page, section and chapter, is done with "p.", "sec." and "chap." respectively, e.g. [1, p. 55], [1, chap. 2], etc.

Prerequisites

The intended readers of this report are people with knowledge equivalent of a 9th semester Software Engineering student, with a basic understanding of parallel programming, and knowledge of computational accelerators and Graphics Processing Unit (GPU)s. A basic understanding of the C# is recommended, as many of the code examples in this report are written in C#. In addition, a basic understanding of the .NET framework is recommended since we are developing a .NET library.

General-Purpose computations on Graphics Processing Units (GPGPU) experience is not required prior to reading this report, since Section 2.1 will describe the basic theory related to GPGPU programming.

Terms

The following list defines how some of the most important terms should be understood.

GPU refers to Graphics Processing Unit, a specialized microprocessor that accelerates graphics computations.

Concurrency refers to computations done simultaneously, but can be interleaved.

Parallel refers to computations that are done simultaneously and in parallel.

GPGPU refers to General-Purpose computing on Graphics Processing Units, i.e. utilizing graphics cards for non-graphical computations.

Device refers to a GPU and its associated memory.

Device code refers to code which is executed on a device.

Host refers to the host system containing the Central Processing Unit (CPU) and main memory.

Host code refers to code which is executed on the host.

Benchmark is a task to be performed a number of times, e.g. a task could be Vector Addition or Matrix Multiplication.

Benchmark suite is a collection of benchmarks.

Benchmark implementation is a specific implementation of a benchmark.

Iteration is a single execution of the task given by the benchmark.

Benchmark run refers to running all the iterations of a benchmark both in steady state and start up.

Run of a benchmark suite is a benchmark run of all the benchmarks in the suite.

Arithmetic Intensity is defined as the number of operations performed per element of memory transferred.

Enclosed CD and Summary

On the enclosed CD the source code developed during this project is available, along with the benchmark results and this report in PDF format. In addition, we have included a summary of the whole report in Appendix F.

Thanks

We will like to give special thanks to Lone Leth Thomsen, our supervisor the past four semesters, for her tireless feedback and guidance. Furthermore we will thank Aage Sørensen for ordering and installing the graphics card we used during this semester, and the IKT-board for sponsoring the graphics card.

Signatures

[illegible]

. . . .

Søren Alsbjerg Hørup

[illegible]

. . . .

Søren Andreas Juul

[illegible]

. . . .

Henrik Holtegaard Larsen

Contents

2 Introduction

GPGPU is the technique of utilizing GPUs for general computations, that is, solving problems other than the graphics rendering problem. During the last decade GPGPU programming has become increasingly accessible, through better support from GPU manufacturers, e.g. Nvidia has released its own programming language, abstract GPU architecture and framework, known as Compute Unified Device Architecture (CUDA), for Nvidia GPUs. Nvidias programming language, CUDA C, has helped remove some of the “black art” of GPGPU programming, but it still requires that the developer knows fundamental GPGPU heuristics, such as which memory types to use in which situations, and how threads should be grouped to maximize performance. CUDA and GPGPU programming is described in more detail in Section 2.1.

9th Semester Project During our 9th semester, we conducted an analysis of the topic of GPGPU where we looked at a concrete GPU architecture, namely the CUDA enabled G80 architecture featured in the Tesla C870 graphic card, and how GPUs can be programmed using GPGPU programming languages such as BrookGPU, CUDA C and Open Computing Language (OpenCL). Furthermore, we gained experience with GPGPU by implementing a ray tracer using CUDA C, and implementing the Boids flocking simulation of one million actors using OpenCL, CUDA C and Brook+.

Additionally, benchmarks were carried out and found that the ray tracer achieved 8 times speedup on the GPU compared to the CPU implementation, and that the flocking simulation achieved 25 times speedup. Finally, we looked at the future trends of GPGPU and found that Nvidia is pushing the GPGPU boundaries by introducing language support for function pointers and object orientation, which were not possible on the G80 architecture. We also found that GPGPU is gaining momentum which is evident by the number of companies which are investing programmer hours in implementing GPU support into their applications, and is evident on the number of GPGPU publications by the scientific community.

Based upon the knowledge and GPGPU programming experience gained last semester, we conclude that GPGPU programing is still in its early stages and requires maturation before it will be accepted by most programmers, but that

GPGPU programming is a skill worth learning if higher arithmetic performance is needed. We also saw last semester that not all types of problems can be solved faster by using the GPU, and even if all problems were effectively solvable using GPUs, the majority of programmers and companies might not be interested in high performance, but instead be interested in e.g. high productivity. Therefore, GPGPU programming is primarily interesting for programmers concerned with performance, as is the case with parallel programming for the CPU.

Motivation Parallel programming on the CPU has existed for several years by utilizing threads, and has been regarded as a "black art" since concurrency faults are common. The same can be said of GPGPU programming, since GPUs are seldom programmed using high level abstractions but instead using lower level abstractions, such as CUDA C or OpenCL C. This means that programmers need to use an Application Programming Interface (API)s such as the CUDA Runtime API or OpenCL API, and languages, such as the CUDA C language or OpenCL C language to move computations from the CPU to the GPU.

Researchers and companies have introduced higher level GPGPU abstractions for well know programming languages, such as the the Accelerator project for .NET [51], which gives programmers the possibility of moving computations to the GPU using their language of choice. The interaction between the CPU part of the application and the GPU part of the application involves invoking functions and marshalling data to and from the GPU. When using one of the higher level abstractions, data between the two parts are often marshalled by a runtime system, instead of manually marshalled by the programmer, and the GPU part of the application is written in the same language, or a subset thereof, as the CPU part.

At the time of writing, CUDA Runtime API, CUDA Driver API and OpenCL API bindings exist for the .NET platform, thus giving .NET programs the ability to invoke GPU functions written in CUDA C or OpenCL C. Also, some solutions such as GPU.NET, allow the GPU part of the program to be implemented directly in C#. GPU.NET does however not provide much abstraction, thus the programmer must still divide the threads into thread-blocks, and write the GPU part of the program as if it was written in CUDA C, albeit with a different syntax. CUDA concepts, such as thread-blocks, will be covered in more detail in Section 2.1.

10th Semester Project In this project, we want to increase the abstraction level of writing GPU powered .NET applications, which means that the programmer must be able to write GPU functions directly in a .NET language, such as C#, while not having to worry about marshalling of data to and from the GPU. Currently, auto parallelization is possible on the CPU by using the .NET Task Parallel Library (TPL) which is part of the .NET 4.0 framework. We want to introduce a similar library but for the GPU, dubbed Accelerated Parallel Library (APL). This allows programmers with experience in using the TPL to port their parallel applications to the GPU, by changing only a few lines of code. TPL supports both task parallelism and data parallelism. In this project, we will focus on the data parallel part of the TPL.

2.1 GPGPU Programming

The aim of this section is to briefly introduce the concept of GPGPU programming, how GPGPU programming is done on todays GPUs, and some of the needed terminology to understand later sections of this report.

There exist several frameworks and GPU specific languages for developing GPU accelerated applications, such as Brook, CUDA, OpenCL and Accelerator. This section mainly introduces CUDA and OpenCL, which we analyzed in our 9th semester project [16].

First, we will briefly cover how GPGPU programming was done in the past using shaders. Afterwards, we will cover the CUDA compute model, which is the model used by Nvidia's GPUs.

Lastly, we will briefly cover the OpenCL compute model, along with the OpenCL C language used to program this model, and compare this model with the CUDA model.

2.1.1 General Purpose Programming Using Graphics API

Prior to abstractions such as CUDA and OpenCL, programmers used graphics APIs such as DirectX and OpenGL to execute custom made programs on the GPU, called shaders.

The graphics API was initialized prior to invocation of the shader, which means that the graphics pipeline is configured to render a square at a given resolution, which is positioned such that it fills the entire screen. The number of threads spawned on the GPU depends on this resolution, e.g. rendering a quad at 1024x1024 resolution means that about a million threads are spawned. [12]

Each spawned thread executes the programmer defined shader, thus allowing highly parallel algorithms to run on the GPU, without these algorithm being graphical in nature. The problems with this approach is that shader programming is very limited, e.g. a shader could not write to an arbitrary address prior to the introduction of compute shaders in DirectX 11[53], and algorithms has to be expressed in a graphical way, i.e. by using the graphics API.

Projects, such as Accelerator which is briefly covered in Section 2.2.6, abstracts away from this graphical nature.

2.1.2 CUDA

CUDA is a parallel computing architecture developed by Nvidia for Nvidia based GPUs [47, sec. 1.2]. CUDA resembles the model presented above, except that kernels, which are equivalent to shaders in DirectX and OpenGL terminology, are much more powerful, e.g. a CUDA kernel is able to randomly access memory and arbitrary branching is allowed.

The programmer writes the kernel in a language such as CUDA C, and is later compiled to a language called Parallel Thread Execution (PTX), the programmer can then issues a kernel call from the host in order to run the kernel on the device. PTX is covered in more detail in Section 3.7. The kernel call determines how many instances of the kernel, i.e. how many threads, should be executed concurrently on the GPU. Each thread executing a kernel instance has a unique thread ID, this can for example be used to determine which piece of the input data the thread should work on. Threads are further grouped into thread blocks and thread blocks are grouped into a grid. This is shown on Figure 2.1. [47, Chap. 2.2]

The CUDA model exposes several types of memory: Global memory, which is shared between all threads, shared memory, which is shared by all threads in a thread block, and local memory, which is private to each thread. This is also depicted on Figure 2.1. Furthermore, some read only memory exists: Constant and texture memory, which can be used by the programmer to optimize certain areas of the kernel, e.g. texture memory can be used to optimize read only parts of the kernel since texture memory is cached on chip memory. When memory from the host is needed on the device it must explicitly be copied back and forth between the host and the device. [47, Chap. 2.3]

With regards to execution, CUDA devices execute threads in lockstep of 32 threads at a time and this grouping is referred to as a warp. Since threads are executed in lock step, branching is not supported. This means that a thread which do not take a branch, is inactive until the other threads in the warp, which did take the branch, returns from that branch. This can result in 1/32th the performance in the worst case, if all 32 threads follow 32 different execution paths. [47, Chap. 4.1]

GPUs which support CUDA have one of several compute capabilities. Each compute capability increases the range of features which is supported on the device, e.g. compute capability 1.1 supports all features of 1.0 but does also support atomic operations [47, G.1]. In addition, two major version of compute capability exists, namely 1.x and 2.x. 2.x introduces many new features, such as 64bit addressing and a common address space which gives better support for pointers. The difference between the two major versions are covered in more detail in Appendix E.

2.1.3 OpenCL

OpenCL is more or less the open version of CUDA, i.e. OpenCL is developed and implemented by many vendors whereas CUDA is implemented by Nvidia. OpenCL provides a programming model which is very similar to the CUDA model, and a language, OpenCL C, which is very similar to CUDA C. The difference between the two abstractions is more or less the naming conventions used. Table 2.1 shows the name translation between CUDA and OpenCL terminologies. Furthermore CUDA is more powerful in some aspects, as described in Section 3.5.2.

Since OpenCL is an open standard, multiple vendors are invited to improve the standard by introducing new features. New features, however, have to be accepted into the standard by the other vendors, whereas Nvidia is the only company responsible for the CUDA standard.

CUDA	OpenCL	Description
Thread	Work-item	One instance of a kernel
Thread block	Work-group	A set of threads
Grid	N-Dimension range	Grouping of thread blocks
Global memory	Global memory	Accessible for all threads
Shared memory	Local memory	Shared between threads in thread blocks
Local memory	Private memory	Private to each thread
Constant	Constant	Read only memory

Table 2.1: Terminology in CUDA and OpenCL

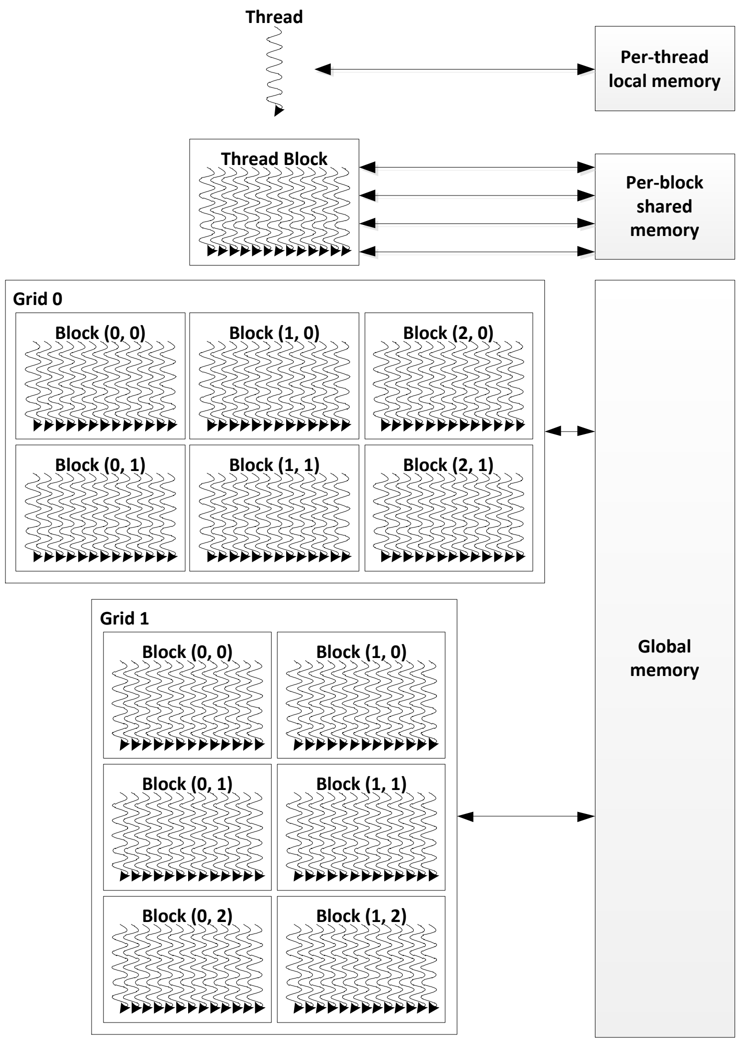


Figure 2.1: Shows the relations between threads, thread blocks, grids and memory types [47]

2.2 Prior Works

In this section we will look at publications in the area of GPGPU programming. Also, we will look at publications dealing with code generation from one language to another within the context of Common Language Infrastructure (CLI), which is the specification of the .NET framework. The aim of this section is to give us inspirations for how to implement APL, by looking on other contributions.

2.2.1 A Parallel Dynamic Compiler for CIL Bytecode

[4] from 2008 deals with the problem of implementing a parallel dynamic translator and optimizer, i.e. a Just-In-Time (JIT) compiler that compiles and performs optimizations in parallel, called Intermediate Language Distributed Just In Time (ILDJIT), which takes as input a Common Intermediate Language (CIL) representation of the source program and translates this representation to native instructions of the target architecture there is then executed. .NET programs are mostly written in a high level language, such as C#, and compiled to CIL instructions, which are then executed on the target architecture using Virtual Machine (VM)s such as ILDJIT. [4, sec. 2]

[4] have designed ILDJIT such that different CIL methods can be compiled simultaneously. Also, instead of directly translating CIL code to the target instruction set, the CIL code is first translated to an Intermediate Representation (IR), and then optimized at runtime by a separate process, i.e. the optimization can also be done in parallel.

According to [4], it is important to define a suitable translation granularity size since a too small translation granularity makes it harder to perform optimizations, while a too big granularity increases the risk of translating code that will never be executed. Also, a small granularity size makes it harder to reason about what state information must be saved, i.e. local variables, thus increasing the overhead of execution. The granularity size chosen by [4] is the size of a method, since a method defines a state boundary, i.e. local state can be destroyed upon exiting a method.

The translation from CIL to executable native code is grouped into five stages in the so called translation pipeline, showed in Figure 2.2. The first stage of the translation pipeline translates the stack based CIL code to register based IR code. The second stage optimizes the IR code. The third stage translates the IR code to the native instruction set of the target architecture, specifically x86 instructions. The fourth stage dispatches the code needed to initialize static memory used by the program, such as static class members, thus making sure that all static memory is allocated prior to use. The fifth and final stage is the actual execution of the generated code on the native architecture. [4, sec. 3]

2.2.2 hiCUDA: High-Level GPGPU Programming

hiCUDA, published in 2009, is an high-level directive-based language for writing applications for CUDA enabled GPUs, which makes it easier to port sequential C code with minimal effort and at performance comparable to handwritten CUDA C code. [14, sec. 1]

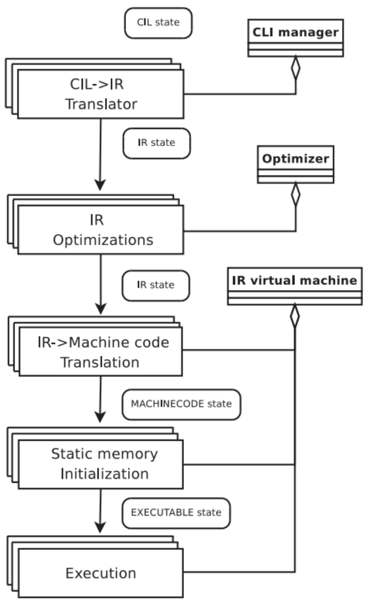


Figure 2.2: Translation pipeline [4, sec. 3]

Given a sequential C program,
`#pragma` directives can be issued that tell the hiCUDA compiler where and how kernel code should be generated. An example of a hiCUDA directive is
`#pragma hicuda kernel matrixMul tblock(4,2) thread(16,16)` and
`#pragma hicuda kernel_end`, which tell the hiCUDA compiler that a kernel with the name "matrixMul" should be created, and that the kernel should be executed with 4∗2 thread blocks containing 16∗16 threads. The C code between the two directives makes up the kernel. [14, sec. 3]

Loops must be parallelized using directives such as
`#pragma hicuda loop_partition over_tblock _over_thread`, since hiCUDA does not provide any auto parallelization of loops to the GPU. This particular directive tell the hiCUDA compiler to output kernel code that loops over blocks and thread. As with loop directives, hiCUDA also supports data allocation directives that allocates GPU memory and data write and read directives that tells the hiCUDA compiler that it must output CUDA C code that either reads or writes data to or from device memory. [14, sec. 4]

2.2.3 GPU.NET

GPU.NET is a proprietary solution that gives developers the ability to write high performance GPU applications, targeting CUDA enabled GPUs, in any .NET supported language, such as C# and F#. TidePowered, the company behind GPU.NET, was founded in 2009 and have since worked on GPU.NET. GPU.NET is currently supported on Windows XP, Vista and Windows 7 and .NET 4.0 is required. [54]

Developers write device methods directly in their .NET language of choice, and specify through .NET attributes that the method should be executed on the GPU, and not the CPU. The .NET method can afterwards be invoked. Prior to invoking of the method, the number of thread blocks and threads are specified by the developer. Data to and from the CPU and GPU is automatically marshalled. [57]

Although GPU.NET removes much of the data marshalling burden from the developers, the developer must still think in CUDA terms, e.g. the kernel code must still be written with block threads and thread indices in mind. Also, GPU.NET only supports primitive types such as integers, and array types; objects and structs are currently not supported. TidePowered says the GPU.NET compiles directly to device code, we assume this means PTX, and not to OpenCL C or CUDA C. [55]

As of February 9 2011, GPU.NET official supports C# and CUDA Software Development Kit (SDK) 3.0. F# and VB.NET can be used, but is not officially supported. Not all opcodes have been implemented, meaning much of the functionality found in CIL is not yet supported. [56]

2.2.4 OpenMP to GPGPU

[20] describes a compiler framework for automatically transforming OpenMP applications to applications that make uses of CUDA enabled GPUs. OpenMP is a platform for shared-memory computations and allows for synchronization between all threads, while the CUDA model only supports synchronization between threads in the same thread block. This problem is solved by exiting the current kernel and starting a new at each synchronization point.

The core contributions of the publication is a source to source translator for OpenMP to CUDA C and several compile-time techniques to improve access to global memory both for applications with regular and irregular memory access patterns.

The resulting compiler framework is evaluated against OpenMP benchmark applications, both with regular and irregular memory access patterns. The evaluation finds the optimizations to improve performance 12x on average over the unoptimized translations.

In 2010, two of the authors of [20] published [19] which extends the OpenMP API with new set of directives to "*control important CUDA-related parameters and optimizations*" [19, sec. 1]. [19] states that by utilizing these directives the developer can tune the application, without detailed knowledge of the CUDA memory model. They also develop tools to assist performance tuning, e.g. a *search space pruner* and a *configuration generator* which can create the different configuration variations produced by the *search space pruner*.

The *search space pruner* reduces the search space for the best configuration by more than 90% thus making it easier to achieve good performance. When the user assists with optimization the performance is up to 88% of that achieved with hand written code, though lower on average.

2.2.5 CUDA DBMS

A Software 10 master thesis from 2009 which deals with the problem of moving computations to a CUDA enabled GPU, by using abstractions commonly found in databases. They proposed a custom .NET Language Integrated Query

(LINQ) provider, which support common database operations such as SELECT, JOIN, and DELETE, thus turning the GPU into a flexible database where data can be inserted, deleted and selected upon. [6, sec. 2.2]

While their benchmarks show that selection queries on the GPU database generally perform worse compared to selection queries on a normal Structured Query Language (SQL) server, it is clear that their GPU database outperforms the SQL server when performing math operations. [6, sec. 5.4]

[6] proposes as future development, to dynamically compile and execute their CUDA kernels instead of Ahead-Of-Time (AOT) compiling each kernel beforehand. At present, an expression is converted to several small kernels that are invoked. They propose to JIT compile the expression to fewer kernels thus taking advantage of register caching and shared memory. Their initial experiments show that such kernels perform worse when the input is small, but perform better than the smaller kernels when the input size increases. [6, sec. 6.6.4]

2.2.6 Accelerator

[52] from 2006 describes Accelerator, which is a library using data parallel abstractions to program GPUs. The Accelerator library is implemented in C# and exposes a data parallel array type to the programmer, along with a number of operations which can be performed on these data parallel arrays. The operations are executed on the GPU using DirectX shaders. Access to individual elements of a data parallel array is not possible without first performing an explicit conversion to a normal array.

To improve the performance of the library, operations performed on data parallel arrays are not immediately executed on the GPU, as this would introduce a significant overhead from running multiple shaders in separate calls. Instead, the operations are stored in a Directed Acyclic Graph (DAG) and only executed when the data parallel array is converted back to a normal array. Another advantages of using a DAG to store the operations are that these can be transformed prior to execution. Transformations can improve the performance of the operations and overcome some of the limitations of shaders, such as the limited maximum size of a shader.

[52] shows that the performance of Accelerator is usually within 50% of that of a hand written shaders, and are often many times faster than a C++ implementation running on the CPU. There are however cases in which Accelerator is significantly slower than the an equivalent C++ implementation, even though a handwritten shader is faster than the C++ implementations. [52] speculates that this is caused by non uniform memory accesses which are handled poorly in Accelerator.

2.2.7 Inspiration

We have looked at prior work ranging from a parallel JIT compiler to a library aimed at accelerating arithmetic evaluations by moving these to the GPU. Some of the publications presented above have given us inspiration which we can use in this project. Looking at the Accelerator project, we see that the abstraction level provided is that of parallel arrays, where the programmer calls methods on the parallel array which is at some point executed on the GPU. We find this abstraction limiting, since the programmer has to express his computations through operations on the parallel array, and convert back and forth between parallel and non-parallel arrays.

We find that the multi-stage pipeline presented in Section 2.2.1 provides a great architecture for us to use: Translate a source language to an IR, optimize the generated IR, translate the optimized IR to native instructions, and afterwards, execute the native instructions. Also, we find that the compile granularity size of a method to be a good size, since TPL deals with `Action` delegates which encompasses one method, with the ability to call other methods, as will be described in Section 3.1.

The GPU.NET in Section 2.2.3 translates CIL code to device code, which are later executed on the GPU, but can prior to the execution be transformed to increase performance. In addition, translating from CIL code allows us to support .NET programming languages other than, e.g. C#.

2.3 Problem Formulation

Our main problem is as follows:

Is it possible to create a library similar to the Task Parallel Library from .NET 4.0, where the data parallel operations are executed on a many-core GPU instead of a multi-core CPU?

To answer this we have created a list of sub-questions, which will help us solve the main problem. We have dubbed our library APL, to distinguish it from the TPL.

- How can source code be accessed at runtime, compile-time or in any other way?
  In order to execute .NET code, the code in question must be accessible, e.g. through pre-processing or at runtime

- How is device code best generated from .NET?
  There are different possibilities for generating device code at runtime or compile-time

- Which optimizations are important when generating code?
  When generating device code, it may be necessary to optimize the code to achieve a speed up

- How does device code generated by our solution fare against hand written device code?
  Since performance is the primary concern of programmers utilizing GPUs for general purpose computations, benchmarks will be carried out to evaluate the performance of our solution.

- How does APL fare against the TPL?
  Moving computations from the CPU to the GPU should increase performance due to the higher theoretical performance of the GPU.

2.3.1 Learning Goals

Beside the questions in the problem formulation, we have created a list of learning goals in the areas we want to gain knowledge during this project.

- Gain knowledge of the .NET framework and its architecture
  Large parts of this project is concerned with .NET and its architecture, and knowledge of these will help us when designing our solution

- Gain knowledge of the Nvidia CUDA and OpenCL compiler and toolchain
  In order to generate GPU code it is useful to know how Nvidias official compiler functions, and possibly find parts of it that can help us optimize the outputted code or be used for the actual compilation

- Gain knowledge of the new features of Compute Capability 2.0
  Compute capability 2.x was not analyzed in our 9th semester project, as we did not have a graphics card of this capability at our disposal. We have now acquired such a card, and the added features may require that we optimize in ways not necessary for compute capability 1.x

- Gain experience with code translation
  During this project we wish to gain further experience with code generation. We have previous had courses in this area which focused on language design and compiler implementation. This semester we want to focus on translation from one language to another

2.3.2 Delimitations

During our previous semester we looked at several GPGPU languages, however, in this project we will only focus on CUDA and OpenCL as these are the most general and getting a lot of attention. Furthermore, we will only be focusing on languages common to CLI, and not other languages such as Java and C, and do not consider the security aspects of our solution.

We have also made delimitations with regards to the memory available to devices, i.e. some problems might require more memory than available on a device, thus requiring some sort of virtual memory solution. We consider this problem outside the scope of this project.

In addition, GPUs tend not to implement 100% IEEE compliant floating point arithmetics, which might influence the precision of the benchmarks. This is also beyond the scope of this project.

Lastly, we will only focus on the `System.Threading.Tasks.Parallel` class of TPL since this is the class which contains the methods for executing parallel computations.

2.4 Development Method

We will be using an incremental method, inspired by agile methods such as Scrum and XP, to develop APL. By splitting our development up in increments, we can have an implemented and working subset by the end of each increment.

We will start this project by conducting an analysis of a number of different solution strategies, which can be used to implement APL. We will choose the strategy based upon pros and cons that we have found during the analysis phase.

We will also create a benchmarks suite such that we can experiment with different optimization techniques, reason about the effectiveness of these techniques, and reason about the performance of our library compared to the TPL found in .NET 4.0, sequential for-loops and CUDA C.

The benchmark suite will also be used as test cases for each increment of our development, so that we are always working towards getting a new benchmark to run without compromising our ability to run existing benchmarks. This means that we must have a wide variety of benchmarks, ranging from very simple benchmarks, such as running an empty loop, to complex benchmarks, where branching and arithmetic computations are carried out. This also means that the benchmark suite should be capable of running automatically, such that we can easily run the whole benchmark suite when a new increment of APL is complete.

2.4.1 Benchmark Approach

There are many things to consider when benchmarking a system. Therefore, we have investigated some of these in Appendix B. Based on this investigation, we have chosen to use the approach as described in this section.

One goal of this project is to see a performance increase when moving computations from the CPU to the GPU, therefore, we want to compare the performance of TPL and sequential for-loop implementations with APL. To determine how well APL fare against other GPGPU abstractions, we will also compare the performance of APL with hand written CUDA C.

Other publications on the subject of GPGPU compares their solutions against the code examples found in the CUDA SDK [14]. Therefore, we will use these as our CUDA C implementation of the benchmarks.

We have been unable to find a benchmark suite which uses TPL and sequential for-loop, with the same benchmarks as those found in the CUDA SDK. Therefore, we will implement the TPL and sequential for-loop benchmarks ourselves.

We only have access to a single compute capability 2.x GPU, so we will only be testing on a single hardware platform.

We will be using the .NET 4.0 framework and the Common Language Runtime (CLR) as these are at the time of writing are the newest versions of Microsoft's official CLI implementation.

We will be measuring the startup performance of the CLR over 40 iterations, to do this we will run each iteration of the benchmark in its own CLR instance. As the first iteration may involve loading data from the hard disk which is later cached, the first iteration will be ignored.

To measure the performance at a steady state, we will continue to run iterations of a benchmark in a single CLR invocation, until a steady state is reached for the past 20 iterations. To take into account that different steady states might occur, this is done two times.

For both startup and steady state measurements, we will be computing a confidence interval at confidence level 99%, in order to reason about the performance of alternatives. If two non-overlapping confidence intervals are found, we will conclude that there is a difference between the two.

As we have limited time, we will only try to limit bias in our benchmarks by trying to create a varied benchmark suite. This is primarily because trying to identify all the factors that might cause bias in a GPGPU powered application would be very time consuming and beyond the scope of this project. Furthermore, causal analysis is hard to perform as it can not be automatized and requires an in-depth analysis of the factors which might influence performance. We will therefore limit our causal analysis to the optimizations, and talk about which improvements might influence the performance of APL.

Since we will primarily be basing our benchmark suite on the CUDA SDK, the benchmark suite might be biased towards the GPU. We do however not see this as a big problem as it has already been shown that there are situations in which the CPU outperforms the GPU and vice versa[21], and that the APL should only be used for the type of problems which performs well on a GPU.

3 Analysis

Our aim is to move computations from the CPU to the GPU using the same abstractions provided by TPL. To achieve this goal, several topics needs to be analyzed before we can create a solution. This analysis is covered in this chapter.

The TPL will be covered in some detail in Section 3.1, with emphasis on how to use the library to program multi-core CPUs within the .NET framework. This analysis will serve as an introduction to the abstraction provided by TPL. The specification of the .NET framework is analyzed in greater detail in Section 3.2, where both the runtime component and the language specifications are covered in more detail.

Moving computations from the CPU to the GPU requires that we translate CPU code to a GPU program, such that it can be executed on the GPU. In Section 3.3, we analyze our options with regards to which source language we should choose.

The device compilers, i.e. the compiler used to compiling code for the device, which is available to developers might have an impact on the choice of target language. Thus, device compilers are analyzed in Section 3.4. The choice of target language is analyzed further in Section 3.5 where the target language is chosen.

Two in depth analysis of the source and target language are carried out in Section 3.6 and Section 3.7, followed by analysis and evaluation of how to translate the chosen source language to the chosen target language in Section 3.8.

3.1 Data Parallelism Using Task Parallel Library

One aim of this project is to implement a library similar to the TPL, but where computations are carried out on GPU and not the CPU. This section will introduce TPL and how it is used, specifically, we will cover the functionality defined within the `System.Threading.Task.Parallel` class as this is the part of TPL which run parallel computations. The section is based on [33].

The .NET 4.0 implementation of the `System.Threading.Tasks.Parallel` class has three methods: `For`, `ForEach` and `Invoke` each with a number of overloads. The `For` and `ForEach` methods deals with data parallelism while

```
// Sequential
int[] output  = new int[100];
for(int x = 0; x < output.Length; x++)
{
  output[x] = x*10;
}

// Parallel
Parallel.For(0, output.Length, delegate(int x)
{
  output[x] = x*10;
});
```

Figure 3.1: Sequential for loop and Parallel.For

```
int out = 0;
Parallel.For(0, 100, delegate(int x)
{
  out += x;
});
```

Figure 3.2: Parallel.For with race condition

`Invoke` deals with task parallelism. In the following, we will cover each of these three methods in more detail.

All overloads of `For` takes at least three arguments, where two are integers defining an index range, and a `Action` delegate defining an action to be executed in parallel once for each index. An `Action` can be a named or unnamed function or an expression lambda. Example usage of the `For` method is shown in Figure 3.1, where an inline `delegate` has been created and given as argument to the `For` method.

The library does not handle race conditions for the user, e.g. the code shown on Figure 3.2 will cause a race condition, since several threads will try to add to the same variable. Thus, the programmer still have to handle this kind of conditions manually.

The `Parallel.ForEach` method can be used to loop over all the elements in an object that implements the `IEnumerable` interface, such as an integer array, and executing an action in parallel for each element. An example of the `Parallel.ForEach` loop is shown in Figure 3.3.

```
int[] indexes = new int[]{0,1,2,3,4,5};
int[] output  = new int[6];
Parallel.ForEach(indexes, delegate(int i)
{
  output[i] = i*10;
}
```

Figure 3.3: Example of Parallel.ForEach

Both `Parallel.For` and `Parallel.ForEach` has overloaded methods. Some of the overloads allow for monitoring and manipulation of the `For` and `ForEach` loop, while some enable local variables for each thread and introduces a final `Action`, which can for example be used to sum up the results of each local variable. This is shown in Figure 3.4. [28] [34]

```
long result = 0;
Parallel.For<long>(0, 100,
  // The initial state of local variable
  delegate(){return 0;},
  // Add i to local variable of the thread
  delegate(int i, ParallelLoopState loop, long local)
  {local += i; return local;},
  // Sum local variables of threads
  // Using the atomic Add method of Interlocked
  delegate(long x){Interlocked.Add(ref result, x);}
);
```

Figure 3.4: Parallel.For with thread local variables

The last method of the `Parallel` class is `Invoke`, which simply executes a series of actions, possibly in parallel, and can thus be used for task parallelism. This is shown in Figure 3.5.

```
Parallel.Invoke(
  delegate(){Console.WriteLine("Action␣1");},
  delegate(){Console.WriteLine("Action␣2");}
);
```

Figure 3.5: Example of Parallel.Invoke

3.1.1 Summary

In summary, the `Parallel` class of TPL contains three overloaded methods, `For`, `ForEach` and `Invoke`, which can be used to do computations in parallel. Each of the methods take one or more `Action`'s as input. An `Action` is a delegate type which can either be a named method, anonymous method or an expression lambda.

We have chosen to only focus on implementing the `For` method since `For` and `ForEach` are very similar in nature and we do not think that it would be efficient use of our time to implementing both of these. We will not implement the `Invoke` method since this focuses on task parallelism which is very poorly supported by GPUs, and while it is possible to create a system which allows task parallelism [13], this is beyond the scope of this project.

3.2 Common Language Infrastructure

The CLI standard describes the overall architecture of CLI and provides a normative description (requirements, recommendations and statements) of the Common Type System (CTS), Virtual Execution System (VES) and Common Language Specification (CLS). The standard also contains the specification of the CIL instruction set, which is conceptually similar to Java byte-code, and a description of the common CLI libraries. [10]

The .NET Framework is a concrete implementation of the CLI specification, which is a specification developed primarily by Microsoft. Other CLI implementations exist, such as the Mono Runtime [40] and Portable.NET [9]. The .NET Framework is a software framework developed by Microsoft targeting the Microsoft Windows operating system. It consists of two main components: the CLR and the .NET Framework class library. [32]

The CLR provides an abstraction layer over the operating system, while the class library provides commonly used constructs. The .NET Framework allows programmers to write programs for a variety of devices, such as desktop computers, mobile phones, servers and clouds, using the same set of tools. These tools include official Microsoft supported programming languages, such as C#, VB.NET, F# and managed C++, the Visual Studio 2010 Integrated Development Environment (IDE) and the .NET Framework library. The current version of .NET is 4.0. [24]

Common Type System

The CTS defines how types are declared and managed by the VES. A type is either a value type or a reference type. VES supports five categories of types: classes, structures, enumerations, interfaces and delegates. Classes, interfaces and delegates are reference types, while enumerations and structures are value types. [26]

Virtual Execution System

The VES provides an environment for executing CIL code, which is the Intermediate Language (IL) code used by the VES, and can be viewed as a virtual machine similar to the Java virtual machine. The CLR is a concrete VES implementation providing an execution environment for code expressed in CIL, which is the target of the official Microsoft supported languages. The VES JIT compiles CIL instructions to native binary code, which are then executed on the given computer architecture. This is depicted on Figure 3.6. Other execution models are also supported, such as interpreting CIL code, and executing native code for legacy or compatibility purposes. The VES includes direct support for 15 data types, such as signed integer, unsigned integer and floating point types, these can be manipulated directly by CIL instructions. [17, Part 1. sec. 12]

Common Language Specification

Programmers seldom write directly in CIL, as it is a low level language, but instead write code using higher level languages such as C# and VB.NET, which are compiled to CIL code. This means that multiple high-level languages can be used when writing a program, i.e. one programmer might write the application core in C#, while another programmer writes the Graphical User Interface (GUI) part of the application in VB.NET.

For a language to be compliant with the CLI, it must be compliant with the CLS. The CLS defines a common set of features that all CLI languages have in common, e.g. Int32 is a supported primitive in the language, all names

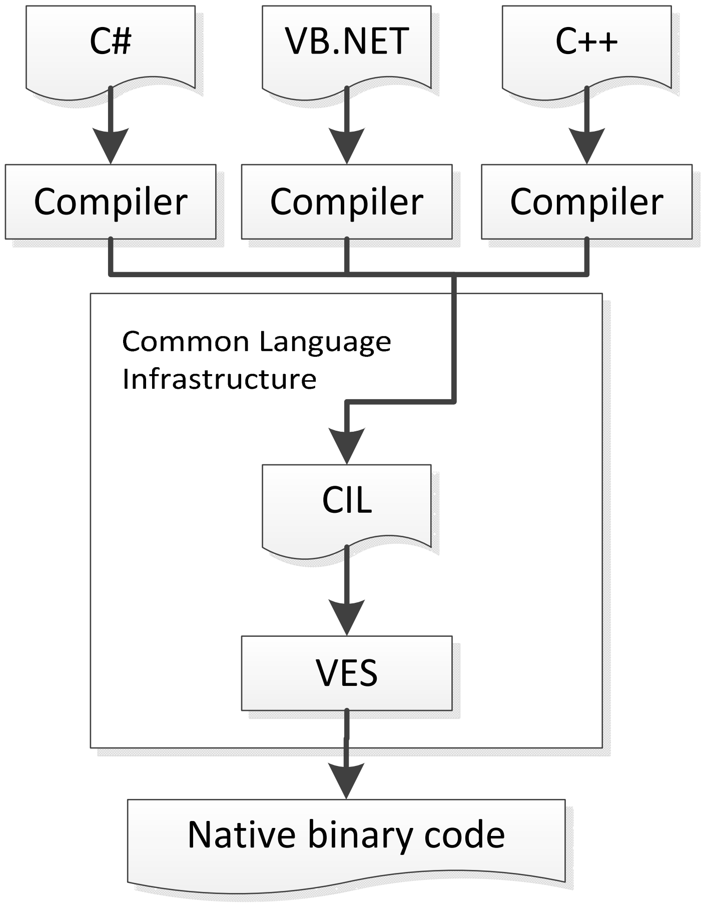


Figure 3.6: Conceptual overview of the Common Language Infrastructure. Illustration from [58].

within a scope are distinct and a class must inherit from the `System.Object` class. Examples of non-CLS compliant features are: Data and function pointers, operator overloading, variable-length argument lists in methods and unsigned types. Some of these features are supported by CLI languages, e.g. pointers and unsigned types are part of C# but not CLS compliant, meaning that using such features might break compatibility with other CLI languages which do not include support for these features. [36]

Code compiled with a compiler targeting CIL is called managed code, not to be confused with managed memory. Managed code benefits from features such as [25]:

Cross-language integration, i.e. classes, methods and data can be shared between two distinct programming languages hereby increasing code reuse.

Cross-language exception handling, i.e. exceptions handling between code written in distinct programming languages.

Enhanced security, i.e. managed code is passed through a verification process before it is executed unless administrator privileges have been granted.

CLI Metadata

CLI metadata is used to describe the structure of code, such as classes and methods. This allows the CLI to locate and load the classes at runtime, resolve method invocations, enforce security constraints, among other things. A CLI component, such as a class and its implementation, carries component metadata. Component metadata describes things such as class declarations, method declarations, where the CIL code of the methods are located, and references to other components. Since metadata is packed within the CLI component, the CLI component is said to be self-described. CLI components are packed together into an assembly which can be deployed. [17, Part 1. sec. 9.1].

CLI metadata is extensible by using CLI attributes. An attribute can be added to all types of metadata objects, such as methods, classes, fields, functions, properties, etc. An attribute contains information that can be accessed at runtime by the VES. An example of an attribute is the "Serializable" attribute, which can be used to denote which classes, and their members, can be serialized. Attributes are language independent and accessible from any CLS compliant language, and can be accessed at runtime by using reflection. Metadata is contained within tables, some of these will be described in more detail in Section 3.6.1. [17, Part 1. sec. 9.7]

Assembly

An assembly is a collection of modules and other resources, such as icons, that provide a set of functionality. A module is a single file containing executable resources, e.g. a C# source file is compiled to a single module. An assembly can contain both public and private resources. Public resources are exposed to and can be consumed by other assemblies, e.g. an assembly might expose a type which can be used by another assembly. Private resources are internal to the assembly, and cannot be consumed by other assemblies. Each assembly has one manifest which declares which files make up the assembly, what types are exposed to other assemblies, and a list of dependent assemblies that must be resolved prior to using the assembly. An application is a type of assembly that exposes an entry point, such as a main function which is commonly found in console application on both Windows and Linux, or Windows services' `OnStart` method, which is called when the service is started. [17, Part 1. sec. 9.5]

3.2.1 Summary

In this section, we have looked at the CLI specification and have found that the .NET framework is a concrete implementation of CLI, while the CLR is a concrete implementation of the VES. In addition, we have covered some of the design goals of the CLI, have found that CIL code is executed on the VES, and that the structure of an assembly is described using metadata. CIL will be covered in more detail in Section 3.6.

3.3 Source Language

In order to move computations from the CPU to the GPU, we have to translate the CPU program to a device program, which means we must translate some source language to a target language. This section will deal with the aspect of which source language we should translate from.

Through the analysis in Section 3.2, we found that we can translate from a high-level CLI language, such as C#, or translate from the low-level CIL code that the high-level CLI languages use as compile target. In addition, we can choose to perform this translation AOT or JIT.

In this section, the approaches will be discussed and evaluated, and a source language will be chosen.

3.3.1 High-level Source Parsing

The first approach is to parse the source file at compile-time, e.g. the C# file containing the method is parsed and device code is generated. This requires that we choose a specific CLI language, such as C#, and this means that our solution is tied to this specific high-level language. This also means that we must write, or adapt, a high-level language parser which is able to detect code used by the APL and output e.g. CUDA C kernel code which can be executed on the device.

By focusing on a CLI language, we can utilize language constructs that are not CLS complaint, e.g. operator overloading and pointers can be utilized if C# is chosen as the high-level language. This however also means that we are tied to one language, and that we more or less void one of the properties of CLI, namely that it supports multiple high-level languages.

Another problem with this approach is that the body of a APL construct cannot reference types or methods where the high-level source code is not available, since device code is generated from the high-level source. This can be problematic, especially when referencing methods within the standard library, since these are typically already compiled to CIL, thus no high-level source code is available. A solution to this problem is to substitute references to the standard library with references to our own library implementation.

Pros

- Writing the solution with a high-level language in mind, such as C#, allows us to utilize language constructs that are not CLS compliant, e.g. operator overloading and pointers.

Cons

- The solution is language dependent and will not work directly with other high-level CLI languages.

- Whenever a type or method is used within a APL construct, where the high-level source code is not available, it has to be replaced by a semantically equivalent implementation where the high-level source code is present.

- The standard library has to be re-implemented or adapted from an open-source implementation, in the high-level language, such that device code can be generated.
- A more complex parser is needed

3.3.2 Low-level Source Parsing

The second approach is to parse the actual CIL code that the high-level language compilers output. This means that such a solution is independent with regards to which high-level language is used to write the code that utilizes the APL. Any CLS compliant language can therefore be used. This can be done AOT, by parsing the output of the high-level language compiler, or it can be done JIT, using reflection and metadata to find and read the CIL code of the code passed to the APL.

This approach does not break with the design goal of CLI, i.e. being able to use multiple languages when writing an application. All CLS compliant types of the standard library, or other libraries, can be used within a APL construct; these do not have to be substituted by our own implementations.

Pros

- All CLS compliant languages are supported.
- Libraries implemented in CIL can still be used, even though the high-level representation is not available.
- If device code is generated at runtime it can potentially be optimized to the specific architecture of the device.

Cons

- Depending on the target language, a device kernel compiler might be required at runtime, which can increase warm-up time.

3.3.3 Ahead of Time vs Just in Time Compiling

An aspect of doing JIT compiling instead of AOT is that the device language compiler must be present on the system in which the application is run. If we choose to target OpenCL C, we get a built in compiler, from the OpenCL API, that can JIT compile and run OpenCL C kernels on the device. If we choose to target CUDA C instead, a CUDA C compiler must be installed on the system. This is not a problem when performing AOT compilation, since only the developer system has to have the compiler tool-chain installed. Furthermore, the compiler may not be intended for JIT compiling, resulting in a slow compiler, ultimately meaning that all speedup gained by running the code on a GPU is lost by the JIT compiling unless some caching is implemented. The analysis and choice of target device language is done in Section 3.5.

Generating device code at runtime, gives the compiler the possibility to optimize the device code for the particular device. As a concrete example, consider a compute capability 1.x device vs a compute capability 2.x device. The compute capability 1.x device supports up to 16K 32-bit registers per Streaming

Multiprocessor (SM), while compute capability 2.x supports up to 32K 32-bit registers per multiprocessor. This means that the compiler should output leaner kernels when targeting the compute capability 1.x device. Although register usage is still a limiting factor on the compute capability 2.x, it is not as big a limitation. Compute capability 2.x is covered in more detail in Appendix E.

3.3.4 Choice

Based on the pros and cons identified above, we choose to use the low-level CIL language as our source language. Especially because this will enable us to generate code from high level CLI languages, thus giving support a several languages without having to implement source parser for each language, but also because it allows us to use libraries without having access to the source code. The low-level language facilitates both JIT and AOT generation kernels by using reflection. With regards to AOT vs JIT compiling of CIL, we will use JIT compiling with the ability to cache compiled kernels, which means that we can contain all required functionality in a library and distribute this library, in the same manner as the Accelerator project, described Section 2.2.6.

3.4 Device Compiler

To help us decide the target device language, this section introduces the OpenCL kernel compiler and the CUDA SDK toolchain, which includes the CUDA kernel compiler and relevant libraries used during compilation of CUDA kernels. As described in Section 2.3.2, we will only focus on these two technologies.

3.4.1 CUDA

The following is based upon [46]. The CUDA SDK toolchain contains several executables: *cudafe*, *filehash*, *nvopencc*, *ptxas*, *fatbin* and *nvcc*. In addition, the toolchain utilizes a C/C++ compiler. This compiler is referred to as *cpp*, which typically is *cl* on Windows and *gcc* on Unix.

Typical CUDA C programs contain both device and host code. Compiling a .cu file can be done by invoking *nvcc*. The device code can be compiled to either PTX code and/or the cubin binary format [47, sec. 3.1].

Typical CUDA C Compilation Run

A parse of a .cu source file, containing both host and device code, involves the following steps:

1. *nvcc* is invoked with the .cu file as input.
2. The .cu file is passed to the platform dependent C/C++ compiler, *cpp*, which performs preprocessing and macro expansion.
3. *cudafe* splits the output of *cpp* into host and device code, such that host code can be compiled to the CPU, and device code can be compiled to the device. *cudefe* also inlines necessary code. *cudafe* produce a .c and a .gpu file, where the .c file contains the host part, and the .gpu file contains the device part.

Extension	**Description**
.cu	Regular host and device source
.cup	Preprocessed CUDA source
.c	C source
.cc, .cxx, cpp	C++ source
.gpu	Gpu intermediate file
.ptx	Ptx intermediate assembly file
.o, .obj	Object file
.a, .lib	Library file
.res	Resource file
.so	Shared object file
.i, .ii	Preprocessed files, i is C, ii is C++

Table 3.1: File types [46]

4. The .gpu file is processed again by *cpp*, then by *cudefe* and once more by *cpp*. These steps produces a .gpu file where dead code analysis and more macro expansions have been performed.

5. The .gpu file is passed to *nvopencc* and *filehash*, described below. *nvopencc* compiles the .gpu file into a .ptx file, which contains PTX instructions that can be compiled to a specific GPU and executed. PTX is described in more detail in Section 3.7.

6. *filehash* produces an application independent device name for the device functions. This name, along with the produced .ptx file, is passed to *fatbin* which produces a .fatbin file that contains the name of the device code and the actual device code.

7. Lastly, the .c file produced by *cudafe*, and the .fatbin file, is passed to *cpp* which combines the two files into a single .c file, which can be compiled and executed by the system.

The file types used can be seen in Table 3.1

nvopencc

The *nvopencc* generates PTX code and is build upon the open64 compiler. It takes a .gpu file as input, containing definitions, such as the definition of dim3 and float3, and the kernel itself at the very bottom.

nvopencc relies on three executables: *gfec*, *inline* and *be*. According to [37], *gfec* is used to produce code in an intermediate language called WHIRL, while *inline* is used to inline definitions and *be* is used to optimize and produce PTX code. *nvopencc* is open source and available at `ftp://download.nvidia.com/CUDAOpen64`, the three executables mentioned above are also in the *nvopencc* releases.

Runtime API and Driver API

CUDA exposes two APIs which can be used by application developers, the Runtime API and the Driver API. The Runtime API is commonly used by application developers. The Runtime API is used by all CUDA C applications.

The Driver API exposes handles that can be used to query and manipulate device state, e.g. `CUdevice` is a handle to a CUDA enabled device, such as `CUcontext`, which is an handle to a a device context similar to a CPU process. Handles allow more control over the CUDA enabled GPU, compared to the relatively higher abstraction level provided by the Runtime API. In addition, the Driver API can load kernels written in the PTX language, which means we can use the Driver API to load and JIT compile our own PTX kernels.

JIT compiling of CUDA C code can be done by using *nvcc* at runtime, with the CUDA C file, and afterwards load the resulting .ptx file at runtime. The *nvcc* may, however, not be suited for JIT compiling, as it is somewhat slow, e.g. our testing shows that compiling the simple seven lined vector addition kernel to PTX code, takes around 0.8 seconds wallclock time on a Linux system, which in some cases is more than a sequential execution of the same problem. The kernel can be seen in Appendix A. In this example, all speedup may be lost by compiling the kernel. Furthermore it requires that the CUDA SDK is distributed to the end user.

3.4.2 OpenCL

The API exposed by OpenCL is conceptually identical to the CUDA Driver API, and allows JIT compiling of OpenCL C kernel programs. This stands in contrast to CUDA C, which does not officially support JIT compiling of CUDA C kernels but only supports JIT compiling of PTX.

3.5 Target Language

Moving computations from the CPU to the GPU, means that we must output device code that can be executed on the GPU. The aim of this section is to choose the target language that will be generated by APL from our CIL based source, and executed on the device.

Recall from the introduction in Chapter 2, that there exist several programming languages, techniques and frameworks that allow GPGPU programming, such as BrookGPU, CUDA C and OpenCL C. Recall from Section 2.3.2 that we have decided to look only at OpenCL C and CUDA C, as these are the most prominent. Also, as described in Section 3.4, CUDA C programs can be compiled to PTX code that can be loaded at runtime and JIT compiled for a specific GPU, such as G80 based GPUs, and executed as a kernel program.

Thus, we find that we have three candidates with regards to which target language we should choose for our solution, i.e. OpenCL C, CUDA C and PTX.

3.5.1 CUDA C as Target Language

The first possible solution is to generate CUDA C code from our CIL code, and then compile the CUDA C code using the CUDA SDK toolchain. Doing this using JIT compilation can be troublesome, since the CUDA Driver API does not support JIT compilation of CUDA C kernels, but only PTX kernels. Thus, the CUDA C compiler must be available if JIT compilation of CUDA C kernels is needed.

CUDA C prior to March 2011 supported classes, as long as the member functions were not virtual, i.e. it was not possible to overwrite an inherited function [47, sec. D.6]. Virtual functions are however present in version 4.0 of the CUDA SDK, which was released as a release candidate in March 2011.

A solution using AOT compilation of CUDA C to PTX simply requires that the resulting PTX code is embedded into the application which uses APL. This is already supported for C and C++ applications, as described in [47, sec. 3.1.4].

Translating CIL code to CUDA C means that we are translating from low-level code to high-level code. According to one publisher [7], translating from a low-level language to a high-level languages can be done by following several steps, such as data flow analysis, control flow analysis and type analysis. Data flow analysis aims to recover high-level expressions, function definitions, parameters for functions, etc. while removing hardware references from the code, such as registers and stack references. Control flow analysis aims to recover conditional statements, loops and nesting levels from low-level branching instructions. Type analysis aims to recover typing information of variables and function return types of functions. This type of translation is also commonly referred to as decompiling. According to [7, sec. 1], several techniques exist for decompiling, but many of these are proprietary and not released to the scientific community, therefore it might be hard to find an appropriate technique for translating CIL to CUDA C.

Translating CIL to CUDA C should however be relatively easy with regards to typing information, since types are saved in metadata which can be referenced by our translator. Metadata also contains much of the information required for data flow analysis, since method name, parameters, return values, etc. are kept within metadata. Control-flow information is not contained within metadata, meaning that CUDA C control flow statements must be generated based on the branching instructions in the CIL code.

3.5.2 OpenCL C as Target Language

The second possible solution is to output an OpenCL C program, similar to CUDA C.

The expressive power of OpenCL C is however lower than CUDA C, since OpenCL C does not provide function pointers and virtual methods. In addition, OpenCL C does not support recursive functions, which have been introduced into CUDA C with compute capability 2.0.

Compiling OpenCL C kernels is done through the OpenCL API, similar to how PTX programs can be compiled through the CUDA Driver API. Thus, choosing OpenCL C over CUDA C allows us to perform JIT compilation without the need to distribute a separate OpenCL C compiler.

Also, choosing OpenCL C makes our solution much less platform dependent than choosing CUDA C, since our solution can be used both on CUDA enabled GPUs, OpenCL enabled GPUs from ATI and Nvidia, OpenCL enabled CPUs, and more.

3.5.3 PTX as Target Language

Recall from Section 3.4 that *nvcc* can output PTX code directly from CUDA C kernels, which are later JIT compiled to the target GPU architecture, or further

AOT compiled to binary form and embedded into the application.

Instead of using *nvcc*, we can translate CIL code to PTX code and let the CUDA Driver API compile the PTX code to the particular CUDA enabled GPU. This means that we have to write a translator from the low-level language of CIL to the low-level language of PTX, thus we do not have to perform the same level of data-, flow- and type-analysis, that we had to perform if we were translating from CIL to OpenCL C or CUDA C.

However, we do not know how much the PTX JIT compiler optimizes the code, which means that we might have to implement more optimizations when outputting PTX code, compared to outputting CUDA C code. This is because we can leverage the existing optimizations done by *nvcc*, when outputting CUDA C code, compared to manually optimizing the generated PTX code. Optimizations such as using shared memory, when appropriate, needs to implemented by both solutions, since *nvcc* does not automatically optimize the use of this and thus has to be manually managed by the programmer.

Outputting PTX also increases our expressive power, since PTX supports features that appear to be non-accessible through CUDA C, such as timing registers [45, sec. 11.2.1.3].

3.5.4 Choice

Based upon the pros and cons presented above, we choose PTX as target language.

If we had chosen to output CUDA C kernels from our APL library, we must have used *nvcc* to generate PTX code from the CUDA C kernels, since neither the Driver API nor Runtime API supports JIT compilation of CUDA C kernels directly. This is in contrast with the OpenCL API, which supports JIT compilation of OpenCL C code, i.e. OpenCL C kernels can be embedded directly as a string and JIT compiled using the OpenCL API. It is possible to JIT compile CUDA C kernels by distributing the Nvidia CUDA compiler.

Translating from low-level representation to a high-level representation require that we perform some kind of data-, control-flow- and type analysis, and is commonly referred to as decompiling. Even though the metadata associated with the CIL code contains much information concerning the types and methods used, control-flow analysis is still a required step before a high-level representation can be generated.

Translating from CIL to PTX is much more straightforward, as we are dealing with two assembly level languages and we can perform a somewhat one-to-one mapping between the two languages, thus voiding the need to perform decompilation, which according to [7] is a complex problem. Also, the choice of PTX removes the problem concerning distribution of *nvcc* and increases the expressive power compared to OpenCL C. The choice of PTX as target does however requires that we learn an additional low-level language and perform low-level optimizations.

3.6 Common Intermediate Language

The aim of this section is to introduce CIL in more detail, since we must translate from stack-based CIL instructions to PTX instruction in APL. We will specifically cover the concept of metadata tables and the evaluation stack, both are important parts of the VES.

3.6.1 Metadata Tables

Recall from Section 3.2, that metadata is used to describe the structure of programs, such as methods, classes, references to objects, and much more. Metadata is stored in one of severals tables, within modules, and is loaded at runtime when needed. Metadata records in metadata tables are referenced by metadata tokens, which are conceptually similar to pointers.

Metadata tokens are four bytes long, where the most significant byte indicates one of the many metadata tables. The other three bytes represents an index within that particular table. Thus, a metadata token points to a specific record within a specific metadata table. [17, Part 2. sec. 22]

An example of a metadata tables is the `TypeDef` table, where each row contains metadata on a type, that includes references to other metadata records containing information such as the name of the type, the name of the namespace of the type, the fields defined within the type, and the list of methods of the type. Another example is the `MethodDef` table, where each row contains metadata on a method owned by a type.

Codeexample 3.1 shows C# code and its CIL counterpart. The `TestMethod` is compiled to a `nop` instruction, i.e. an instruction that does nothing, a `call` instruction, i.e. an instruction that calls a method, and a `ret` instruction, i.e. a return instruction. The `call` instruction takes as operand a 4 byte metadata token, 0x06 0x00 0x00 0x02, which points to a metadata record in a metadata table, specifically to the second row of the MethodDef table. The VES is able to locate and invoke the CIL code of the `TestMethod2` method, since the metadata record pointed to by the metadata token, $0x06000002$, has a signature column that points to an area in memory containing the CIL instructions of `TestMethod2`.

Given a metadata token, reflection can be used to retrieve the particular method, field, type signature, etc. from a module.

Specifically, the `Module.ResolveMethod(int)` method returns a reference to an object of type `MethodBase`, while `Module.ResolveField(int)` returns a reference of type `FieldInfo` to a particular field in a class. Afterwards, reflection can be used to invoke the method, retrieve values of fields used by the method, etc. [30]

```
C# code:

public static void TestMethod()
{
  TestMethod();
}
public static void TestMethod2()
{
}

CIL code:
;TestMethod()
00:    nop
01:    call ,0x06000002
02:    ret
```

```
;TestMethod2()
01:    nop
02:    ret
```

Codeexample 3.1: C-sharp code compiled to CIL code, which refer to a MethodDef metadata record

3.6.2 Evaluation Stack

Recall from Section 3.2 that the VES is responsible for the actual execution of a CIL program, and can be viewed as a virtual machine executing CIL instructions. The VES is a stack machine and as such, most of the CIL instructions operate on values on the stack instead of taking operands. Load instructions refer to instructions that load values from the stack, while store instructions refer to instructions that store values on the stack. The `add` instruction is an example of a load and store instruction, that pops two value from the stack, adds them, and pushes the result back to the stack. [17, Part 1. sec. 12.1]

Native data types are also supported and are mapped at runtime, e.g. the integer data type is mapped to a 32bit integer on a 32bit system, and a 64bit integer on a 64bit system. This also means that data type sizes are generally not known at compile time, and must be determined at runtime. Pointer sizes are also determined at runtime unless some explicit datatype is used as a pointer within the CLI, e.g. an unsigned int32 could be used as a pointer but would only work correctly on 32bit systems unless some pointer conversion is in place. Forcing pointer sizes at compile time should be avoided, but can be necessary, e.g. when accessing unmanaged memory. [17, Part 1. sec. 12.1.1]

Object references, having type `O`, and managed pointers, having type `&`, are used within the CLI in a managed the environment. An `O` references is used to reference objects as a whole, e.g. an `O` reference can be used to reference an instance of the class `Foo`. An `&` pointer is used to reference fields within an object, such as a method, member variable, or specific location in an array. Managed pointer can point to locations that are not under the control of the garbage collector, such as static variables and the stack. Since data can be moved around by the garbage collector, object references and managed pointers can be changed during garbage collection. [17, Part 1. sec. 12.1.1.2]

3.6.3 Summary

As described in this section, reflection can be used to fetch metadata from the metadata tables, such as metadata of a particular field of a class and the CIL code of methods. In addition, we have seen that CIL is a stack-oriented language, which uses the evaluation stack to perform evaluations. In Chapter 4, this information is used to implement the IR and the translation from the stack-oriented CIL code to register-oriented PTX code.

3.7 Parallel Thread Execution

PTX is a low level parallel thread execution virtual machine and Instruction Set Architecture (ISA). The aim of PTX is to expose the GPU as a data-parallel computing device, while providing a stable programming model spanning multiple GPU generations, a stable ISA, and efficient execution on Nvidia's GPU architectures. PTX provides a machine independent ISA for compilers to target, such as the CUDA C compiler, which means that future GPUs can change their native architectures without breaking backward capability with older applications. [45, chap. 1.2]

With the release of PTX 2.1, PTX supports features commonly found in virtual machines meant for CPUs, such as stack support for compute capability 2.x targets, indirect branching, indirect function calls, and more. The indirection introduced in PTX 2.1 allows for features such as C++ virtual functions. [45, chap. 1.3]

The aim of this section is to give an overview of PTX. Some of the details of the PTX ISA will be explained in more detail throughout Chapter 4, when needed.

3.7.1 Programming Model

The following is based upon [45, chap. 2].

The programming model of PTX is conceptually identical to that of CUDA C. The GPU is a device which is used as co-processor to the main CPU for compute intensive applications. Several of thousands of threads are executed on the device concurrently, executing the same kernel function. Threads are organized into one or more Cooperative Thread Arrays (CTA), these are conceptually similar to thread blocks in CUDA C and are executed concurrently or in parallel. CTAs are organized into a grid, which is conceptually similar to a grid in CUDA C.

CTAs are either one, two or three dimensional shapes, specified by a three-element vector which specifies the number of threads in each CTA dimension. As with a CTA, grids are also one, two or three dimensional shapes and can be accessed by a three-element vector. In addition, each grid has an identifier, and all threads within a grid execute the same kernel program.

As with CUDA C, PTX supports a number of different memory spaces, such as register memory, shared memory, constant memory and texture memory. The memory spaces are refereed to as state spaces.

3.7.2 Source Format

The following is based upon [45, chap. 4].

PTX programs are human readable text files, similar to assembly code generated by a compiler or disassembler. PTX source files are commonly JIT compiled to the desired GPU architecture, but AOT compiling can also be done.

A PTX statement is either a directive or an instruction, and can be prefixed with an optional label, and ends with a semicolon. `.reg .f32 r1;` is an example of a directive statement that declares a register `r1` of type `.f32`, i.e. a 32bit floating point. `add.f32 r1, r2, 0.5;` is an example of a instruction statement, which adds the floating point value of 0.5 to the value of register `r2`, and stores the result in the `r1` register.

Comments in PTX uses the same syntax as in C/C++, Java and C#, with `//` denoting a one line comment, while `/* */` denotes a multi line comment.

3.7.3 State Spaces

The following is based upon [45, chap. 5.1].

A state space is a storage area, all variables used within a PTX program are stored in a state space, and PTX defines a number of state spaces, such as the `.reg` and the `.global` state spaces.

Register The `.reg` state space describes a register storage area with high bandwidth and low latency memory access. The amount of available registers is limited and register variables will be spilled to the local state space, causing a decrease in performance, if this limit is exceeded. Scalar register sizes are restricted to a width of 1, 8, 16, 32 and 64 bits, while vector registers have a width of 16, 32, 64 or 128 bits.

Special Register The `.sreg` state space contains all special registers, such as the three-element vector that specify the CTA dimensions. All variables in `.sreg` are read only.

Constant The `.const` state space is similar to the constant memory area in CUDA C, thus variables contained within this state space are cached read only variables that are initialized by the host. The `.const` memory space is organized into eleven 64KB banks, i.e. eleven logical units of memory storage. Note that the CUDA C Programming Guide states that CUDA C programs only have access to one 64KB bank [47, sec. G.1], whereas PTX allow all banks to be used [45, chap. 5.1.3].

Global The `.global` state space is similar to global memory in CUDA C. All threads have access to the `.global` state space, and synchronization instructions such as the `bar.sync` must be used to ensure that memory writes have been carried out to avoid race conditions within a CTA. Generally, threads must be able to do their work without relying on the work of other threads, and threads in different CTAs cannot communicate and synchronize their global state space access.

Local The `.local` state space is memory that is local to each thread. Prior to compute capability 2.0, local memory was non-cached memory and had similar performance as global memory. With the introduction of compute capability 2.0, local memory is cache-able. On compute capability 2.x devices which supports call stack and recursive functions, the stack is located in local memory.

Parameter The `.param` state space is used to pass input arguments from the host to the kernel. The `ld.param` instruction can be used to fetch the value contained within a parameter, and store this value in a register. The location of the `.param` state space is implementation specific. With the release of PTX 2.0, the `.param` state space can also be used as parameters in device functions. Prior to PTX 2.0, device functions only supported the `.reg` state meaning that structures that did not fit a single register must be flattened and passed using multiple registers.

Shared The `.shared` state space is shared between all threads within a single CTA. This state space is conceptually similar to shared memory in CUDA C. Memory access can be done when using `.shared` state space, such as the use of a broadcast, i.e. all threads within the CTA reads at the same time and at the same address, thus conserving bandwidth.

Texture The `.tex` state space is global read only memory which is cached on the device, i.e. memory bandwidth consumption when using texture memory is reduced compared to global memory. All threads share the `.tex` state space and typically have support for 128 texture bindings, i.e. up to 128 separate textures can be used at a time. The `.tex` has been deprecated in favor of new types that specify textures. The PTX statement `.tex .u32 tex_a;`, which declares a reference to texture memory, is equivalent to `.global .texref tex_a` which uses the new texture type, `.texref`.

3.7.4 Types

The following is based upon [45, chap. 5.2].

PTX defines several fundamental types of different sizes, which reflect the native data types supported by the target architecture, e.g. the compute capability 1.x and compute capability 2.x architectures. Signed and unsigned integers of different sizes are supported, e.g. `.s32`, `.u16`, `.s8`, `.u64`, etc. where `s` and `u` denotes signed and unsigned integers.

Floating points are also supported, i.e. `.f16`, `.f32` and `.f64`. Thus, PTX can be regarded as a strongly typed language. Untyped "types" are also supported, e.g. `.b32` denotes 32bit untyped data. Fundamental types are compatible if they share the same type and the same size, e.g. it is possible to add two variables of type `.f32` together, but not two variables of type `.f32` and `.s32`. A conversion must be done prior to the addition. One exception to this rule is that unsigned and signed integer types are compatible if they are the same size, i.e. it is possible to add two values of type `.u32` and type `.s32`, and save the result in either a `.u32` or `.s32` register.

The last fundamental type is the predicate type, denoted by `.pred`. `.pred` registers are used as guards in instructions, e.g. `@p add.u32 x,y,z` is only executed if the predicate `@p` is true.

3.7.5 Example

Codeexample 3.3 shows a compiler generated PTX program, from the CUDA C source shown in Codeexample 3.2, that performs an addition of the `a` and `b` array, and stores the result in the `c` array. The kernel uses the thread index as index into the arrays, thus, thread 0 will perform $c[0] = a[0] + b[0]$, while thread i will perform the ith addition, i.e. $c[i] = a[i] + b[i]$. Note that the PTX example is not the direct output from the *nvcc* compiler. It has been cleaned, i.e. unneeded comments have been removed, some variables have been renamed, and white spaces have been added.

The statements on line 1 and line 2 declares that this PTX program is a PTX 1.4 program, i.e. features of PTX 1.4 and below are supported, and that the PTX program supports compute capable devices of 1.0 and higher.

Line 3 declares the `addArray` kernel which takes three parameters. Each parameter denotes an array, and each array is represented as an 32bit unsigned integer which is used as a pointer into the memory area where the elements of the array is kept.

Line 5 – 7 declares three `.u16` registers, ten `.u32` registers and five `.f32` registers. The numbered brackets denotes the number of registers to create, e.g. the statement `%rh<3>` denotes three registers named `%rh1`, `%rh2` and `%rh3`.

Line 8 – 9 converts the thread id stored in `%tid.x` from type `.u16` to type `.s32`, i.e. from an unsigned 16bit integer to a signed 32bit integer, and stores the result in the `%r1` register. Afterwards, the value stored in the `%r1` register is converted to an `.u16` type and stored in the `%rh1` register. The reason for the two conversions are to remain backward compatible with older PTX version, since the `%tid.x` special register was a `.u16` type prior to PTX 2.0, where it is now a `.u32` type. This conversion is not needed if the PTX version was set to 2.0 instead of 1.4 [45, table 115].

Line 10 performs an integer multiplication on the value stored in `%rh1`, with the constant four, and writes the result to the `%r2` register. The `%r2` register is later used as an byte offset into the a, b and c arrays, thus, thread 0 access the first four bytes of the arrays, while thread n accesses the four bytes at index $n*4$. The multiply of four is required since PTX does not support C style array indexing.

Line 11 – 13 loads the address of the first array, stored in the `a` variable, into `%r3` register. Afterwards, the byte offset stored in `%r2` is added to `%r3` and the result is stored in the `%r4` register. `%r4` now holds the address of the of the nth float in the `a` array. The nth value is loaded into the `%f1` register using the `ld.global.f32` instruction at line 13.

Line 14 – 16 is essential the same as line 11 – 13 but on the `b` array. On line 17, the two nth float values of the `a` and `b` arrays are added together and stored in the `%f3` register. Line 18 – 19 calculates the address of the `c` array where to place the result. Line 20 stores the value of `%f3` into the `c` array, at the address stored in `%r8`, using the `st.global.f32` instruction.

Line 21 exits the kernel, marking the end of the computation.

```
__global__ void square_array(float *a, float *b, float *c)
{
  int id = threadIdx.x;
  c[id] = a[id] + b[id];
}
```

Codeexample 3.2: The CUDA C source of the PTX program

```
1   .version 1.4
    .target sm_10
3   .entry addArray (.param .u32 a,.param .u32 b,.param .u32 c)
    {
5   .reg .u16        %rh<3>              ;
    .reg .u32        %r<10>              ;
7   .reg .f32        %f<5>               ;
    cvt.s32.u16      %r1  , %tid.x       ;
9   cvt.u16.u32      %rh1 , %r1          ;
    mul.wide.u16     %r2  , %rh1 , 4     ;
11  ld.param.u32     %r3  , [a]          ;
    add.u32          %r4  ,  %r3 , %r2   ;
13  ld.global.f32    %f1  , [%r4+0]      ;
ld.param.u32     %r5  , [b]          ;
add.u32          %r6  ,  %r5 , %r2   ;
ld.global.f32    %f2  , [%r6+0]      ;
add.f32          %f3  ,  %f1 , %f2   ;
ld.param.u32     %r7  , [c]          ;
add.u32          %r8  , %r7  , %r2   ;
st.global.f32    [%r8+0], %f3        ;
exit;
}
```

Codeexample 3.3: Example of PTX program which sums two array.

3.7.6 Summary

During this section we found that the programming model of PTX is very similar to the CUDA programming model, e.g. threads are grouped in CTAs which are equal to CUDA thread blocks described in Section 2.1.2. PTX also has the same memory spaces as CUDA along with some extra, such as `.sreg` which contains special values, e.g. thread index number and CTA size.

PTX is a low level language similar to assembly languages known for example from x86. PTX is register based, and each registers is declared with a specific type, e.g. `.reg .s32 %r1` declares a signed 32 bit integer register, which can be referred to as `%r1`, while `.reg .f64 %f1` declares a 64 bit float.

3.8 Translation

This section will look at ways of translating CIL instructions to PTX instructions. We start by presenting two methods for generating a IR for the `Action` delegate method, which is passed to APL through the `Parallel.For` method. Next, we will analyze the differences between stack based and register based ILs and how to translate from the former to the latter.

3.8.1 Expression Tree

The `Expression` namespace of the .NET library can be used to build an expression tree which can be traversed to generate PTX code. However, the `Expression` class can only be used to parse Expression Lambdas, i.e. methods that do not use assignment such as the one shown in Codeexample 3.4.

```
delegate int del(int i);
del myDelegate = x => x * x;
```

Codeexample 3.4: Example of Expression Lambda which squares x [29]

The methods of TPL take `Actions` as input, i.e. a delegate type which points to either a lambda expression, a named method or an anonymous method as described in Section 3.1. Even though the `Expression` class can only automatically generate expression trees for lambda expression, expression trees can be created manually using static methods on the `Expression` class. Thus we can implement our own expression tree generator to handle the `Action` delegate type which the `Expression` cannot parse, i.e. named and anonymous methods. [27]

```
Action<int> action = new Action<int>((int x) =>
{
  Console.WriteLine(x);
});
Byte[] ilcode = action.Method.GetMethodBody().GetILAsByteArray();
```

Figure 3.7: Example showing how to get CIL from an `Action`

3.8.2 Reflection

The classes in the `Reflection` namespace can be used to retrieve information about entities [35], such as the `Action` delegate. For example, `Action` has a `Method` property of the type `MethodInfo`, which stems from the `Reflection` namespace. This `MethodInfo` class contains a `GetMethodBody` method which returns a `MethodBody` object, which is also part of the `Reflection` namespace. This `MethodBody` object can be used to retrieve the CIL code of the given `Action` by calling the `GetILAsByteArray` method, this is shown on Figure 3.7.

Likewise, references to data and methods defined outside the action can also be accessed through reflection, as mentioned in Section 3.2. The CIL code can now be parsed and used to generate an IR which can be further optimized and translated to PTX code as described in Section 3.8.3.

3.8.3 CIL to PTX Translation

CIL and PTX are both used to express general purpose programs, although CIL is primarily used to express CPU programs, whereas PTX is primarily used to express GPU kernel programs. The following will give an overview of the differences between the two languages. Next we will look at how the stack based CIL language can be translated to the register based PTX language and which possible problems can arise from this.

Stack Based vs Register Based

CIL is entirely stack based whereas PTX is register based.

Nearly all CIL instructions operate on values stored on the evaluation stack, e.g. the `add` instruction pops two values from the evaluation stack and pushes the sum of the values back to the evaluation stack. Another example is the `ldelem.i4` instruction which pops an array address, pops an index, fetches an `int32` value contained within the array at the index, and pushes this value to the stack as an `int32`. Most instructions use the stack for operands. Some instructions do however take operands, such as the branching instructions which take the address to jump to.

Looking at PTX, nearly all instructions take one or more operands, e.g. the `add` instruction takes a destination operand and two source operands. Arithmetic evaluations are not carried out using a stack, instead, registers are used. Thus, the programmer or compiler declares several registers that can hold the values of computations. Another example is the `ld.global` instruction which is used to fetch the value contained within an array in global memory, by taking the address of the array offset by some index, and the register to hold the value.

```
             reg a1, a2, a3   // Declare registers
push 0x1     mov a1, 0x1      // Push/move data to stack/registers
push 0x2     mov a2, 0x2      // Ditto
add          add a3, a1, a2   // Add data
```

Figure 3.8: Pseudo code for a stack based assembly language to the left, and the translated pseudo code for a register based assembly language to the right, after applying SSA to all push and pop operations

A similar operation in CIL requires that the address and index of the array were pushed to the stack.

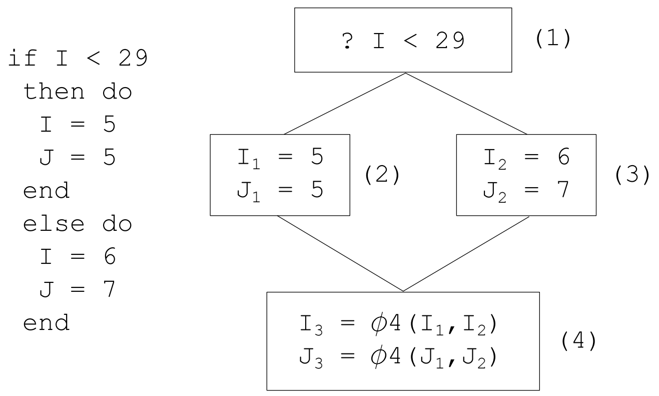
Stack to Register Translation

As we saw above, translating CIL to PTX requires that we translate from a stack based machine to a register based machine. A simple way of doing this is by creating a new register for each instruction which pushes data to the stack, and use this register when an instruction pops data from the stack. This method is called Static Single Assignment (SSA) [1] and is commonly used in compilers to perform optimizations, e.g. identifying redundant writes.

Figure 3.8 show an example of SSA translation from stack based to register based pseudo code. When looking at this pseudo code, we can see that some of the registers created by using SSA could have been left out, e.g. if `add` supports values as argument the registers `a1` and `a2` could be avoided. However, when loading PTX source with the CUDA Driver API, e.g. with the `cuModuleLoadDataEx` method, the level of optimization and the maximum number of registers used can be set [43]. Though the documentation does not explicitly state that these optimizations also optimize the number of registers used, this may be the case, thus the potential overuse of registers may not be an issue.

To test whether the JIT compiler optimizes the register consumption, `cuFuncGetAttribute` from the Driver API can be used. `cuFuncGetAttribute` retrieves information attributes about the JIT compiled PTX code, e.g. number of registers used and the amount of local memory used [42]. We can determine if the JIT compiler removes some of the registers used, by comparing the number of registers used by the JIT compiled PTX code to the number of registers declared in the input PTX file. Registers are spilled to Local memory, if more are used than available. The local memory resides in device memory and thus has the same high latency and low bandwidth as global memory [47, sec. 5.3.2.2], thus eliminating register usage by moving registers to local memory can lead to slowdown. To test this we assess the local memory consumption after the PTX code has been JIT compiled.

We conducted a test on a CUDA C kernel compiled with *nvcc* to PTX, the CUDA C kernel and resulting PTX file is shown in Appendix A. The PTX file declares four unsigned int 16 bit registers, nine unsigned int 32 bit registers and ten unsigned int 64 bit registers. When loading the PTX file with `cuModuleLoadData` and using `cuFuncGetAttribute` to retrieve register count and local memory usage, we see that the register count has been reduced to four registers in the JIT compiled PTX code, and with 0 byte usage in local memory. From this we can conclude that the JIT compiler, at least in some cases, is able to make large reductions in the number of registers used, and that Nvidia's own *nvcc* compiler also has an overproduction of registers, when producing PTX files, compared to the optimized version. Thus, the overuse of registers introduced by SSA may not be a problem and we will not perform any register elimination in our IR.



```
if I < 29
 then do
  I = 5
  J = 5
 end
 else do
  I = 6
  J = 7
 end
```

Figure 3.9: Phi function. From [1]

By using SSA we may face the problem of deciding which variable name to use after branching, e.g. the example in Figure 3.9. The left part of Figure 3.9 shows pseudo code for a program which checks whether `I` is below 29, and if it is, sets `I` and `J` to 5, if not, `I` is set to 6 and `J` is set to 7. The right part of the figure shows a Control Flow Graph (CFG) representation of the left part, with each node representing a basic block, i.e. a portion of code which does not contain jumps within it, and SSA applied to variables. For more about basic block see Section 4.2.4. In to order use `I` and `J` after the branching, caused by the if statement in the CFG, an extra joint point `(4)` is added. This join point decides which of the variables should be used, e.g. I_1 is assigned I_3 if `I` was below 29 and I_2 if `I` was above 29, thus after the join point, I_3 can be used without worrying about which path was chosen. [1]

Through our initial experiments we have not encountered any CIL code where values were not written back to e.g. local variable array as shown in Figure 3.10. Line 2 and 3 pushes 0 to the stack and pops it into local variable 0, line 4 and 5 pushes argument 1 (`isTrue`) to the stack and pushes 0 to the stack. Line 6 compares whether the two elements on the stack are equal and line 7 and 8 stores the result in local variable 1 and loads it back to the stack. Line 9 branches to label `IsTrue` (line 13) if the top element of the stack is non-zero, else continue with subsequent lines. Line 10 and 11 writes 1 to the stack and writes it back to local variable 0 and line 12 jumps to `IsFalse` (line 16). Line 13 and 14 pushes -1 to the stack and writes it local variable 0. Line 17 and 18 loads local variable 0 and replaces the value at field `value` with the value on the stack. Through this result, we see that all temporary values on the stack are written to local variables before the next branch.

However, if we should encounter this problem, we are able to use the ϕ function as described above. One implementation would be to split the ϕ function in the number of arguments it takes, and move it into the basic blocks. For example, J_3 = ϕ4(I_1,I_2) can be split in I_3 = I_1 which is put in basic block

```
nop                            static void Main(bool isTrue)
ldc.i4.0                       {
stloc.0                          int x = 0;
ldarg.0                          if(isTrue)
ldc.i4.0                           x = 1;
ceq                              else
stloc.1                            x = -1;
ldloc.1                          value = x;
brtrue.s IsTrue                }
ldc.i4.1
stloc.0
br.s IsFalse
IsTrue:
ldc.i4.m1
stloc.0
IsFalse:
ldloc.0
stsfld int32 ConsoleApplication1.Program::value
ret
```

Figure 3.10: Example of how computations are copied to local variable array before exiting basic block

`(2)` and $\texttt{I}_3$ `=` $\texttt{I}_2$ which is put in basic block `(3)`.

3.8.4 Summary

In this section, we have looked at two ways of generating our IR, and we have chosen to parse the CIL which can be retrieved using reflection. In addition, we have looked at the problem of translating stack-based instructions to register-based instructions, and have chosen to use SSA to help us in this regard. Lastly, we have looked at a potential problem of using SSA, namely that branches can be problematic with regards to intermediate values. Through our experiments, we saw that intermediate values were saved in a local variables and not on the stack, thus removing the need to implement the ϕ function.

3.9 Summary

In Section 3.1, we described the TPL, which we in Chapter 2 stated that we wanted to move to the device. Furthermore we decided that we would only focus on implementing the `Parallel.For` method in APL as we found the other parts of TPL to be beyond the scope of this project.

In Section 3.2, we described the CLI architecture, and we found that .NET is a concrete implementation of CLI. In addition, we found that the CLR is a concrete implementation of the VES and that the VES executes CIL instructions by JIT compiling these to the underlying computer architecture. We also found that one of the primary design goals of CLI was to facilitate multiple high-level languages.

In Section 3.3, we looked at which source language we should translate from, be it a high-level language such as C#, or the low level language CIL. We found that the best choice was to use CIL as source language, since one of the design goals of CLI is to facilitate multiple languages. CIL was therefore chosen as the source language. Additional we chose to use JIT compiling since this means that APL can be used without making changes to ones existing compiler toolchain.

In Section 3.4, we looked at the official CUDA toolchain found in the CUDA SDK, and saw that we had the possibility to generate PTX code from CUDA C code. We found that *nvcc* is not a good choice for JIT compilation, due to *nvcc* taking nearly a second to compile a very simple kernel, and that *nvcc* would have to be distributed along with APL.

In Section 3.5, we chose the target language among three candidates, PTX, CUDA C and OpenCL C. We found that translating CIL to CUDA C or OpenCL C required that we implemented a decompiler, which would require control-flow analysis, among other things, to construct high-level source code. We found that choosing PTX meant that we were able to produce a closer one-to-one mapping between the source and target language, and that PTX was more powerful. We therefore chose PTX as target language.

In Section 3.6, we analyzed CIL and found that metadata tables can be accessed using reflection, and that the CIL instructions of methods can be fetched. In addition, we found that CIL is a stack-oriented language, thus it uses an evaluation stack to perform the computations.

In Section 3.7, we analyzed PTX and found that the PTX language is a register-oriented language, thereby standing in contrast to CILs stack-orientation. We also found that PTX exposes several state spaces, and that PTX is a typed language.

In Section 3.8, we analyzed how we could translate CIL code to PTX code. We found that we could use reflection to access the CIL code and use SSA to help us translate the stack-based CIL instructions to the register-based PTX instructions, and that even though branches could be problematic with regards to SSA, this was not observed in our initial experiments.

With the analysis complete, we will in Chapter 4 talk about the development of the APL.

4 Development

This chapter covers the development of APL, from requirement to implementation.

The functional and non-functional requirements to APL are described in Section 4.1, where the functional requirements are based upon a number of benchmarks and the non-functional requirements are individually rated.

The design of APL is covered in Section 4.2 where the different classes and their relations are covered. In addition, the IR is covered in more detail in this section.

The `Parser` and `Optimizer` are covered in Section 4.3 and in Section 4.4. The `Emitter` and `Invoker` are covered in Section 4.5 and in Section 4.6. During the development of the four components, five optimizations were found and implemented. These are covered in the section dealing with the particular component where the optimization is implemented.

Lastly, a summary of the development chapter is given in Section 4.7.

4.1 Requirements

This section defines our functional and non-functional requirements for the APL implementation. Recall from Section 2.4 that the functional requirements are based upon a number of benchmarks. The chosen benchmarks are covered in Section 4.1.1. Afterwards, the non-functional requirements are covered in Section 4.1.2.

4.1.1 Functional

In order to run the benchmarks, we need to implement the functionality used by these. Thereby these benchmarks will serve as acceptance tests for each iteration. A brief description of each benchmark, along with their required functionality, is described below. The benchmarks are also covered in more detail in Appendix C.

1. **Overhead Benchmark:** Execute an empty `Action`.
   The first step is to have support for an empty kernel. This benchmark has the following concrete requirements:

   - APL must implement an interface which resembles that of TPL.
   - APL must implement the `APL.Parallel.For` method such that it takes an `Action` along with two integers defining the range that the `For` loop should iterate over.
   - APL must be able to generate an empty device kernel in valid PTX.
   - APL must run the resulting PTX kernel on the device.

2. **Vector Addition Benchmark:** Execute an `Action` that adds each element in two arrays of floating points together, and stores the results in a third array of floating points.
   This step introduces the first arithmetic operation, namely addition, and imposes marshalling requirements to APL.

   - APL must be able extract the CIL code of the `Action` at runtime, and the data utilized by the `Action` must be identified.
   - APL must parse the CIL code and generate an IR.
   - APL must generate PTX code from the IR.
   - APL must marshall the data used by the `Action` to the device before execution, and back to the host after execution.

3. **Matrix Multiplication Benchmark:** Execute an `Action` which multiplies two matrices.
   The third benchmark introduces branches to the kernel, and two dimensional rectangular arrays.

   - APL must support the required branching opcodes, not implemented in the previous benchmark
   - APL must support two dimensional rectangular arrays.

Importance	*Not important*	*Less important*	*Important*	*Very important*
Efficiency				✓
Scalability			✓	
Correctness			✓	
Comprehensibility			✓	
Testability			✓	
Interoperability		✓		
Usability		✓		
Reusability		✓		
Maintainability		✓		
Reliability		✓		
Portability	✓			
Security	✓			

Table 4.1: Nonfunctional requirements

4. **Black Scholes Benchmark:** Execute an `Action` which implements the Black-Scholes financial model.
   The Black-Scholes model is widely used in other benchmark suites, requires function calls, and utilizes many of the mathematical functions found in `System.Math`.

   - APL must support function calls.
   - APL must support the required mathematical functions, such as `Math.Log`.

4.1.2 Non-Functional

The non-functional requirements, and their importance, are specified in Table 4.1. Table 4.1 is used to decide how much focus should be put on each non-functional requirement during development. Each non-functional requirement is explained in more detail in the following paragraphs and is inspired by the non-functional requirements found in [23]. In this context, "Implementation" is defined as APL.

Efficiency - Very Important *The economical utilization of the technical platform's facilities*
In order to achieve good performance, we must utilize the platform as efficiently as possible, e.g. by producing efficient PTX kernel code. In addition, the CUDA model exposes different memory spaces which have different performance characteristics, such as fast on-chip shared memory compared to slower but more abundant global memory.

Efficiency is fulfilled when the APL implementation produces a PTX kernel which, for a given benchmark, achieves the same performance as the kernel produced from the CUDA C implementation. Also, APL should impose minimum overhead on the marshaling and invoking of the kernel, this should be comparable to the CUDA C implementation.

Scalability - Important *How well the implementation scales with regard to the input*
The programmer specifies three inputs to the APL: the program to run on the device, the iteration range used by `For`, and, implicitly, the data which the program operates on.

The first requirement is that APL must translate the CIL program to an equivalent PTX kernel which performance scales equally well or better than the sequential for-loop, TPL and CUDA C implementation.

Also, APL must scale to an arbitrary iteration range size. Different devices impose limits on the number of threads and thread-blocks, which means that there is an upper bound on the size of the iteration range. Circumvention of this restriction is beyond the scope of this project, as described in Section 2.3.2.

With regards to the data size, APL will not impose any restrictions. However, a device might not have enough memory to hold the input data, thus requiring some form of memory swapping between host memory and device memory. This is beyond the scope of this project, as described in Section 2.3.2, and APL requires that all needed data is kept in device memory. In addition, since we only have access to one compute capability 2.0 GPU, we will not implement support for multiple CUDA enabled devices.

Based upon the requirement that the generated PTX kernel must scale at least as well as the other implementations, and that the iteration range and data sizes are bound by the device, we rate scalability as an important requirement in APL. This requirement is fulfilled if the data size is bounded only by the amount of memory available on the device, and that the resulting PTX kernel scales the same as the other implementations.

Correctness - Important *Fulfilling the formalized requirements*
In order to improve the correctness of APL we will be using the benchmarks as acceptance tests. Thus, if the APL implementations of the benchmarks produce the same results as the other implementations described in Appendix C, we have fulfilled correctness.

Comprehensibility - Important *The effort of ensuring a coherent understanding of the system*
Beside facilitating maintainability, segmenting the code will also help increase comprehensibility if each segment has a well defined responsibility along with well written and documented code. However, as we are three developers working closely together, we will not focus greatly on this area. We will however document our solution using UML and describe the implementation through this chapter, thus we rate comprehensibility as important.

Testability - Important *The expense of ensuring that the system fulfills the requirements*
APL will be implicitly tested by our benchmarks, which are also used as acceptance tests as described in Section 4.1.1.

To ease testing we will implement a benchmark runner, as described in Appendix D, which will be used to run all benchmarks for all implementations, output timing result from the benchmark, and verify the benchmarks computation. We find the testability requirement fulfilled when this is implemented.

Interoperability - Less Important *The expense of coupling the system with other systems*
The system should be able to handle the necessary parts of CLI to run the benchmark implementations, but it is not a requirement that it can be coupled with other systems.

Usability - Less Important *Adaption to the technical-, work- and organizational environment*
To increase usability we will use the same names as TPL, thus developers accustomed to TPL will be able to easily adopt APL, however, beside this usability is not priority for this project.

Reusability - Less Important *The ability of using parts of the system in other systems*
Our design should to some extent facilitate reuse, e.g. by dividing the components by responsibility, but this property should not be at the cost of other non-functional requirements rated more important, thus, reusability is rated as less important.

Maintainability - Less Important *The expense of finding and correcting errors in the system*
The system needs to be maintainable during this project, through the iterations and the refactoring that will possibly be required to optimize the code or implement new features, thus we split the code up in segments containing different responsibilities.

Reliability - Less Important *Fulfillment of the required level of precision when solving a task*
The system must be able to handle errors and throwing exceptions, thereby allowing the developer to act on these problems. However, the non standard compliance of floating point math is not considered, and is rated beyond the scope of this project as described in Section 2.3.2. The results of the benchmarks will however be compared to each other.

Portability - Not Important *The expense of moving a system to another technical platform*
We will not be focusing on other languages than CIL and PTX, and we will only target compute capable 2.x GPUs, thus portability is not a priority.

Security - Not Important *Securing the system against unwanted access to its facilities and data*
Security is outside the scope of this project and will not be considered, as described in Section 2.3.2.

4.2 Design

Based upon the functional requirements in Section 4.1, and the architecture described in Section 2.2.1, we have created an UML 2.0 class diagram that depicts the major classes, and their relations, used in APL. This diagram is shown on Figure 4.1. In addition, we have created an activity diagram, which depicts the abstract activities. This gives a better understanding of what activities are present within APL, whenever a `Parallel` method is called. Also, a sequence diagram is covered, which depicts the concrete sequence of operations, when calling the `Parallel.For` method exposed by APL. The aim of the sequence diagram is to concretise the abstract actions into concrete operations. The activity and sequence diagrams are covered in Section 4.2.2 and Section 4.2.3 respectively.

4.2.1 Class Diagram

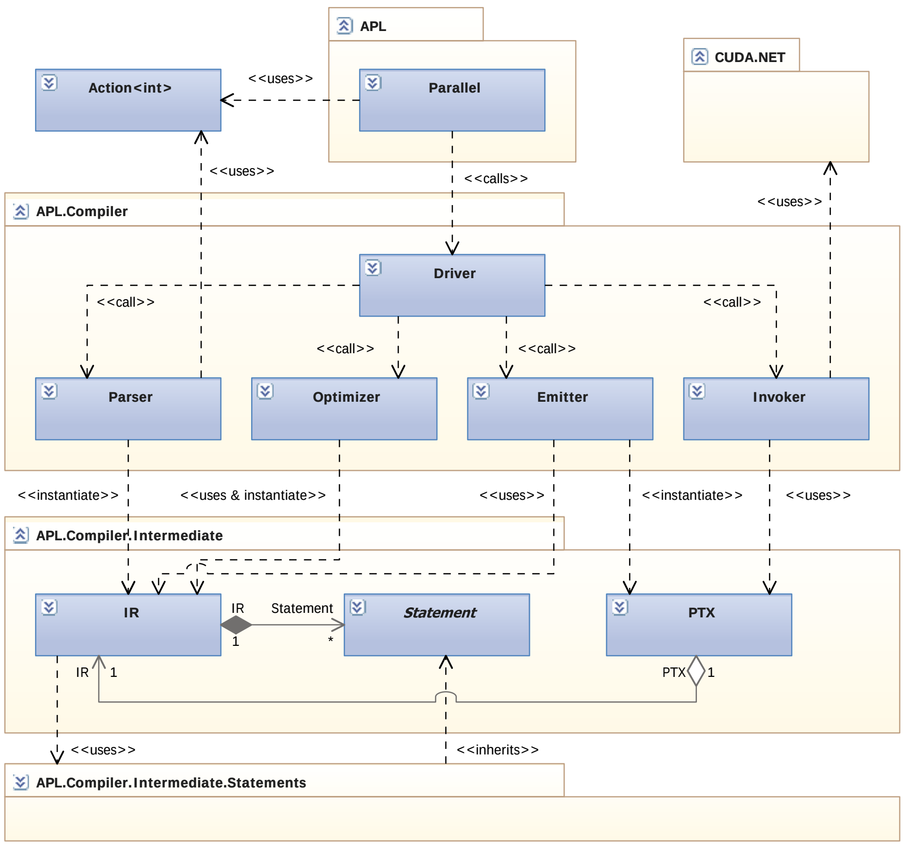


Figure 4.1: Class diagram of APL

APL defines four namespaces:

- `APL`

- `APL.Compiler`
- `APL.Compiler.Intermediate`
- `APL.Compiler.Intermediate.Statements`

These namespaces and their content will be described below.

APL Namespace

The `APL` namespace contains the `Parallel` class, which is the only public available class outside of APL.

Parallel The `APL` namespace contains the `Parallel` class, which has the same signature as the `Parallel` class contained within the TPL. The `Parallel` class contains the static `For` method. This method can be used by the programmer to offload computations, which are contained within an `Action` delegate, to the device.

APL.Compiler Namespace

The `APL.Compiler` namespace, contains the `Driver`, `Parser`, `Optimizer`, `Emitter` and `Invoker` classes. Each class, apart from the `Driver`, is responsible for one step in the compilation process.

Driver The `Driver` moves the compilation process forward, by calling static methods on each of the four classes, and pass each intermediate result from the previous step to the next. In addition, the `Driver` implements the first optimization: a cache which caches the resulting PTX code, thus compilation of the method body referenced by an `Action` object only happens once. We expect that this optimization will increase performance when executing the same `Action`, since compilation can be a time consuming process. We have chosen to cache the whole code, and not individual PTX functions, even though the latter might provide better performance in situations where the same method is used by different APL invocations. The reason for this choice is that function call support is first implemented in later increments.

Parser The `Parser` parses the CIL bytecode of the `Action` object, thereby generating an IR of the CIL code. The Parser is described in Section 4.3.

Optimizer The `Optimizer` performs optimizations on the IR, thereby generating a optimized IR. The Optimizer is described in Section 4.4.

Emitter The `Emitter` traverses the IR and generates PTX instructions. These are packed into a PTX kernel that can later be loaded by the CUDA driver API and executed on the device. The emitter is described in Section 4.5.

Invoker The `Invoker`'s responsibility is to start the PTX kernel on the device, while making sure the required data is marshalled to the device prior to execution, and marshalling the data back when execution has finished. The `Invoker` is described in Section 4.6.

APL.Compiler.Intermediate Namespace

The `APL.Compiler.Intermediate` namespace, contains the `IR` and `PTX` classes, among others. These classes are used by the different steps of the compiler.

IR The `IR` class is the IR of the CIL code contained within the `Action` delegate. It is generated by the `Parser` and modified by the `Optimizer`. The IR, along with the classes used within it, are covered in more detail in Section 4.2.4.

PTX The `PTX` class is a container class that contains the PTX instructions emitted by the `Emitter`. A `PTX` object has a reference to the `IR` object which was used to generate the PTX instructions. This gives the `Invoker` easy access to the metadata through the `IR` references within the `PTX` object.

4.2.2 Activity Diagram

Figure 4.2 shows the activities that are carried out whenever the `Parallel.For` method is called from the APL.

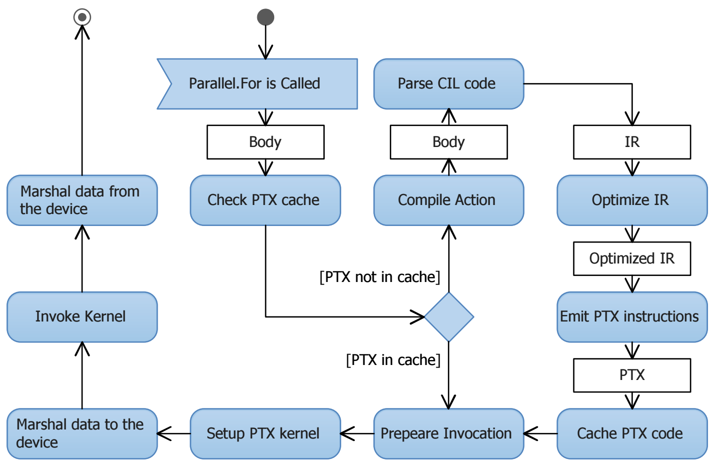


Figure 4.2: Activity diagram of a call to the Parallel.For method

An application calls the `For` method of the `Parallel` class contained within the `APL` namespace of the APL library, this results in the *Parallel.For is Called* activity.

The PTX cache is checked to see if the method body has been compiled to PTX code or not. If it has not, the compilation is started. This is denoted by the *Compile Action* activity.

The CIL code is parsed, the IR is optimized and PTX instructions are emitted and cached such that subsequent calls to `Parallel.For` with the same input does not result in multiple compilations of the same method body.

After the *Cache PTX code* activity, APL proceeds with the preparation of kernel invocation, where the PTX kernel is loaded and the required data is marshalled to the device, such as the fields used by the method body.

Afterwards, the kernel is invoked, and the resulting data is marshalled back from the device to the host. This marks the end of the activity.

In another run, using the same method body, the *Check PTX cache* activity results in the *Prepare Invocation* activity, since an existing PTX version of the method body is located in the cache.

4.2.3 Sequence Diagram

The activity diagram presented in Section 4.2.2 shows the abstract activities that are present when calling the `Parallel.For` method. The sequence diagram in this section cover the concrete operations that are performed when calling the `Parallel.For` method. For better readability the sequence diagram is divided in two parts: Compilation sequence and invocation sequence.

Compilation Sequence

As shown on Figure 4.3, an Application calls the `For` method on the `Parallel` class. The first argument specifies the start index of the for loop, the second argument specifies the end index and the last argument specifies an `Action` delegate containing the body of the for loop.

The `Execute` method of the `Driver` is called with the same arguments. Afterwards, a check is made to see if a `PTX` object corresponding to the `Action` delegate is kept in the cache. If not, the `Parse` method of the `Parser` is called, with the `Action` delegate as input.

The `Parser` parses the CIL code of the `Action` delegate and creates an `IR` object with the parsed CIL statements, as `Statement` objects. The `IR` object is returned to the `Driver` and fed to the `Optimize` method of the `Optimizer`.

The `Optimizer` performs some optimizations on the `IR` object, and the optimized `IR` is returned to the `Driver`, where it is fed to the `Emitter` using the `Emit` method.

The `Emitter` traverses the `IR` and generates PTX instructions. A `PTX` object is created, which contains the generated PTX instructions. The `PTX` object is returned to the `Driver`, where it is cached in the PTX cache using the `CachePTX` method.

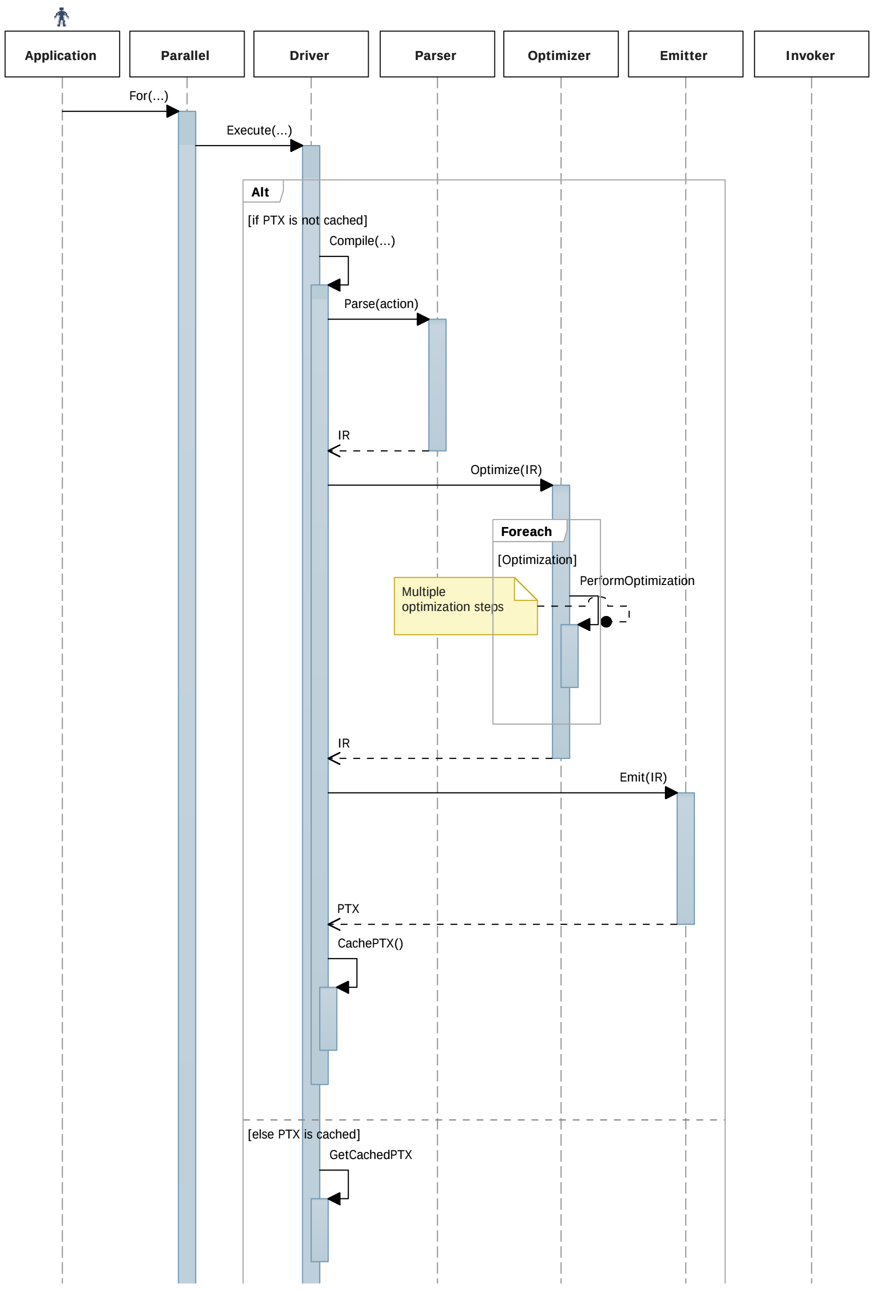


Figure 4.3: Compilation sequence diagram of a call to the Parallel.For method

Invocation Sequence

As shown on Figure 4.4, the `Driver` calls the `Invoke` method of the `Invoker` with the `PTX` object as argument. The `Invoker` loads the kernel using the `LoadKernel` method, marshalls the required data to the device using the `MarshalTo` method, and sets the arguments of the kernel using the `SetArguments` method.

Afterwards, the kernel is invoked using the `InvokeKernel` which blocks until the kernel has finished executing. The data is marshalled back from the device using the `MarshalFrom` method and control is returned to the application which called the `Parallel.For` method.

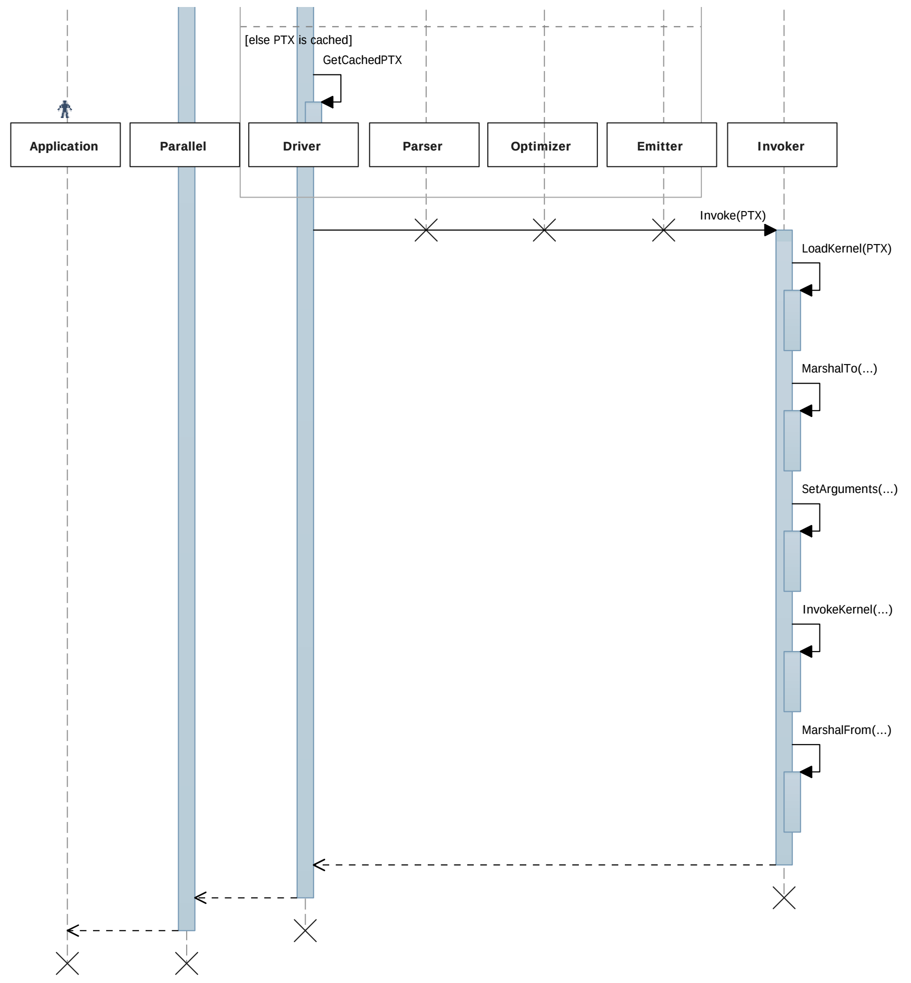


Figure 4.4: Invocation sequence diagram of a call to the Parallel.For method

4.2.4 Intermediate Representation

Recall from Section 3.8 that we have decided to parse and generate an IR from the CIL code using SSA. The IR is transformed by the `Optimizer` in the next step of compilation, to produces an optimized IR, this approach is inspired by Section 2.2.1. The aim of this section is to cover the classes of the IR and their relations. These are shown on Figure 4.5.

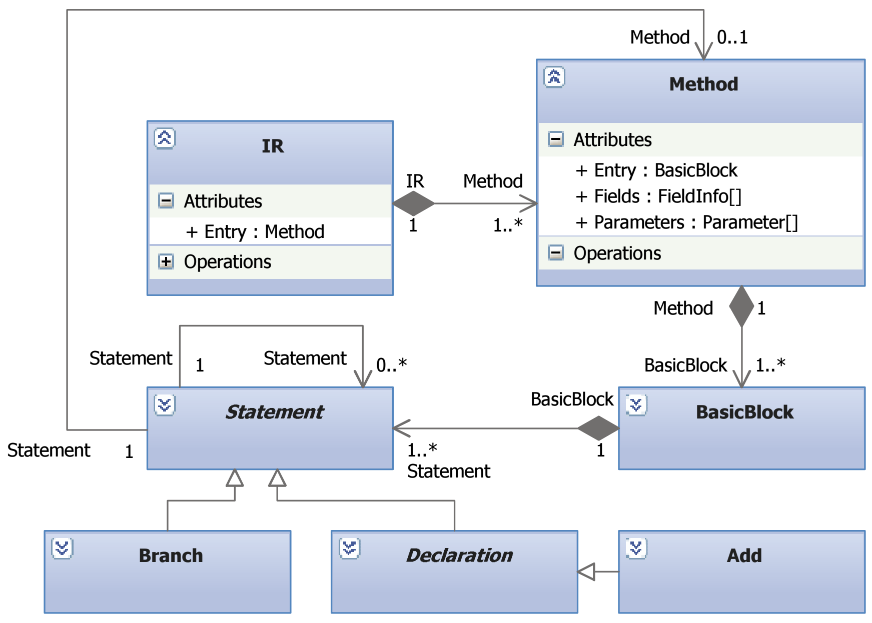


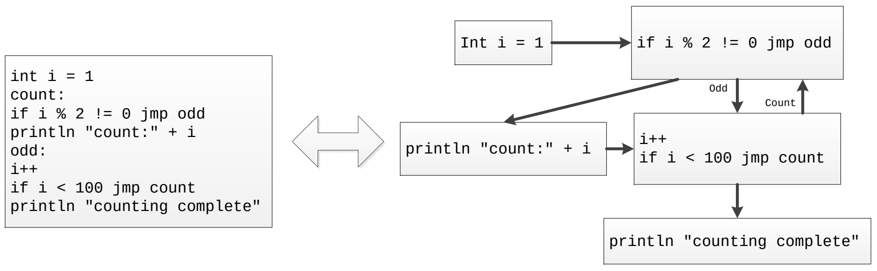
Figure 4.5: The class diagram of the intermediate representation

The IR is composed of four main classes: an `IR` class, a `Method` class, a `BasicBlock` class, and an abstract `Statement` class. In addition, several subtypes of the `Statement` class exists, each representing a specific statement type, e.g. `Add` represents an addition statement, `Call` represents a call statement to a method, etc. Statements that produces a value inherit from the `Declaration` class, this class is described in more detail in Section 4.3.

The `IR` class consists of several `Method` objects, where each `Method` object represents a method. An `IR` object has at least one method, and each `IR` has exactly one entry point, i.e. a method which is used as by PTX kernel entry point, this is explained in more detail in Section 4.5. The entry method can be fetched using the `Entry` property of the `IR` object.

The `Method` class encapsulates all basic blocks that make up a method, and exposes the `BasicBlock` object which is used as entry point into the method through the `Entry` property. The `Method` also exposes the `Fields` and `Parameters` properties, where the `Parameters` property returns the list of parameters used by the method, and the `Fields` property returns the list of fields that are used by the method. The `Fields` and `Parameters` properties are used by the `Emitter` and `Invoker` and their uses are explained in more detail in Section 4.5 and Section 4.6.

The `BasicBlock` class represents one basic block in the method, i.e. an array of statements where the last statement allows branching outside the block and the first statement allows branching into the block.



```
int i = 1
count:
if i % 2 != 0 jmp odd
println "count:" + i
odd:
i++
if i < 100 jmp count
println "counting complete"
```

Figure 4.6: Pseudo code which have been split into basic blocks

As an example of the use of basic blocks, see Figure 4.6. The pseudo code on the left increments the integer variable `i` until it reaches the value of 100. For each increment, if the value is even, the value is printed to the console, and at the end of the count, "counting complete" is printed to the console. This piece of code is subdivided into five basic blocks, where the first initializes the variable `i`, the second jumps if the value of `i` is odd, the third prints the value of `i`, the fourth increments `i` and jumps to the second basic block if the value is less or equal to 100, and the fifth prints the "counting complete" message to the console.

The abstract `Statement` class represents a statement which perform some computation, such as an addition, assignment, call to method, etc. `Statement` objects can reference other statements if they depend on these, e.g. an `Add` statement must reference two `Assignment` statements that contain the name of the variables to add together. Statements can also reference methods, such as a `Call` statement that performs a call to the method. In addition, statements which perform branching can refer to statements in other blocks.

4.2.5 Summary

In this section, we covered the design of APL. We have divided APL into four components: the `Parser`, the `Optimizer`, the `Emitter`, and the `Invoker`. In addition, we have chosen the structure of our IR and are now ready for the implementation of each component.

4.3 Parser

The aim of this section is to describe the `Parser` component of APL in more detail. The `Parser` is responsible for parsing the CIL data and thereby generating an IR. The `Driver` calls the `Parse` method of the `Parser` class, with a `MethodInfo` object as argument. The `MethodInfo` object contains all metadata of a method, such as the return type, parameter types, etc. and contains a reference to the body of the method such that the corresponding CIL code can be accessed. The first method which is parsed is the method referenced by an `Action` delegate, i.e. the delegate which is used by the `For` method of the `Parallel` class. Other referenced methods are also parsed.

4.3.1 Traversal of CIL

The CIL instructions of a method are traversed and `Statement` objects are generated based upon the encountered opcodes, e.g. an opcode of 0x00 generates a `Nop` statement, while an opcode of 0x58 generates an `Add` statement. The generated statements are saved in one or more `BasicBlock` objects, depending on the number of branches in the method, and the `BasicBlock` objects are saved within a `Method` object which represents the parsed method. Recall from Section 4.2.4 that an `IR` object is composed of one or more `Methods` which are composed of one or more `BasicBlocks`, which are composed of one or more `Statements`.

4.3.2 Evaluation Stack

Recall from Section 3.6 that the VES uses an evaluation stack to carry out evaluations, e.g. the `add` instruction pops two values from the stack and pushes their sum to the stack, and that we aim to convert these stack based instructions to register based instructions. As described in Section 3.8.3, this can be done by following the principles of SSA.

All statements inherit from the `Statement` class in some way, either directly or by inherit from the `Declaration` class, which in turn inherits from the `Statement` class. `Declarations` are statements that produce a value, e.g. the `Add` and `Assignment` statements are `Declarations` while the `Nop` and `Branch` statements are not. `Declarations` contain a `Type` member which denotes the type of value produced by the statement, e.g. an `Add` statement might produce a value of type `Int32` or `Float`.

Each time a `Declaration` statement is created and added to a `BasicBlock` by the `Parser`, the same `Declaration` statement is pushed to a stack. The stack is used to keep track of which values other statements consume, e.g. an `Add` statement will pop two `Declarations` from the stack and push itself to the stack. The two `Declarations` represents the values which the `Add` instruction adds together. This stack is therefore used as a compile-time evaluation stack.

Boolean evaluations are handled in a similar manner, except that most boolean instructions have a 16bit opcode instead of an 8bit opcode, e.g. the compare equal (`ceq`) instruction has the opcode $0xFE01$. When a `ceq` instruction is parsed, a `CompareEqual` statement is generated which produces a value of 1 or 0, by at runtime comparing the two `Declarations` prior to it. If the `Declarations` are equal a 1 is produced, and if they are not a 0 is produced.

4.3.3 Branching

Local branching instructions, i.e. branching within a method, such as the unconditional branch instruction `br`, takes one operand which denotes the target address to branch to. In CIL the target is represented as a signed offset from the next instruction, e.g. `br` 0x10 transfers control to the address of the next instruction + 0x10. When parsing a CIL instruction and generating a `Statement` object, the address of the instruction is saved within the `Statement`. This address is later used to calculate the exact `Statement` a branching statement transfers control to.

Conditional branches are handled in a similar manner, e.g. the branch on equal instruction (`beq`) pops two values from the evaluation stack, compares the values, and branches if they are equal.

All branching statements inherit from the `Branch` class, which defines two members: `Target` and `BranchTo`. The `Target` member holds the signed offset as described above, and the `BranchTo` contains the target `Statement` of the `Branch` statement. The `BranchTo` member is computed after all statements belonging to a method have been found, by traversing the list of `Statements` and comparing their `Address` to the computed address of the branch.

This information is later used by the `Emitter` to generate labels in PTX code that the branching statements can point to.

4.3.4 Calling other Methods

When the `Parser` encounters a `call` opcode, a `Call` statement is generated which represents a call to a method. `Call` statements are `Declarations` since a method might return a value. In CIL, a `call` instruction contains a metadata token, which points to the metadata of the given method. This metadata token can be dereferenced using reflection, and the corresponding `MethodInfo` object is saved within the `Call` statement.

The `Parser` is then called recursively on the new method, which produces an `IR` for that particular method. The `IR` of the method is merged with the current `IR` object, i.e. the methods of the newly parsed `IR` object are added to the methods of the current `IR` object. The return type of the `Call` statement is set to the return type of the corresponding method. The merging is done such that only one `IR`, containing all methods, is produced at the end of the parse.

4.3.5 Local Variables, Fields and Method Arguments

Recall from Section 3.8.3 that local variables used within a CIL program are kept in the so-called local variable array, and not on the evaluation stack. CIL instructions such as `ldloc` and `stloc` are used to load and store the content of local variables, kept in the local variable array, to and from the evaluation stack. These type of instructions are parsed and a `LoadLocal` statement is generated, if the instruction is a `ldloc`, while a `StoreLocal` statement is generated, if the instruction is an `stloc`. `LoadLocal` is a `Declaration` since the load produces a value, while a `StoreLocal` is a `Statement` that consumes some other `Declaration`.

Arguments are handled in the same way. Arguments can be loaded to the evaluation stack using the `ldarg` and values can be stored using the `starg` instruction.

Fields are also handled in the same way, where the `ldfld` and `stfld` are used to load and store values of field.

4.3.6 Arrays

In CIL, one dimensional arrays can be accessed using instructions such as the `ldelem` instruction, which loads an array element at the specified index onto the evaluation stack, the `stelem` instruction, which pops a value from the evaluation stack and stores this in the array at a specified index, and the `ldlen` instruction, which pushes the length of the array onto the stack.

To access an element in the array, the CIL program pushes the metadata token of the array to the evaluation stack, along with the index of the element to access. In case of a `stelem` instruction, the value of the store operation is also pushed prior to the execution of the `stelem` instruction. The `Parser` generates a `LoadElement` or `StoreElement` statement, if a `ldelem` or a `stelem` instruction is encountered. The `LoadElement` statement is a `Declaration` since the `ldelem` instruction produces a value, i.e. the value contained in the array at the particular index.

Rectangular arrays, i.e. multi dimensional arrays where each dimension has a fixed size, do not have native support within CIL. Instead, these are implemented using `SetValue` and `GetValue` member methods of the `Array` class. The `GetValue` method takes two integers as arguments that represent the index of each dimension. The `SetValue` method takes in addition to the index, an argument containing the value in which to set. Instead of translating the rectangular access to method calls, we parse the call and create a `SetElementRect` statement, in case of a `SetValue` call, or a `GetElementRect` statement, in case of a `GetValue` call. Thus, we make a single statement instead of translating the opcodes of `GetValue` and `SetValue` for multidimensional arrays, in our `IR`. By doing this we avoid the need to implement support for objects.

4.3.7 Math Functions

Mathematical functions, such as the `Abs`, `Log`, `Sqrt`, etc. found in the `System.Math` class is required by one of the benchmarks. However, many of the methods are not implemented in CIL code, which is evident when performing reflection on these methods, meaning that we are not able to translate calls to these methods directly. These methods are instead implemented in the VES.

Instead, whenever a `call` opcode is encountered by the `Parser` to one of the `System.Math` functions, a statement which reflects the particular method is generated, e.g. a call to `System.Math.Abs` results in a `Abs` statement instead of a `Call` statement. The `Emitter` is afterwards able to generate the required PTX code for the math functions.

4.4 Optimizer

The `IR` produced by the `Parser` can be optimized by running it through the `Optimizer`, which performs transformations on the `IR`. We have decided to implement two optimizations in this class, namely the fused multiply add optimization and the optimization where only fields which have been written to are marshalled back from the device, and fields which are read by the device, are marshalled to the device prior to kernel invocation. These optimizations have been chosen sine we know that copying data between the device and host is a slow process, and that *nvcc* produces code that makes much use of multiply add.

We refer to the last optimization as "Copy Omit", since unnecessary copies are omitted. These optimizations, and what we expect from them, are described below.

4.4.1 Fused Multiply Add

An expression containing both a multiplication and addition, such as $1 * 2 + 3$, is parsed such that two statements are generated, namely one `Multiply` and one `Addition` statement. When generating PTX code, these statements will result in two PTX instructions, one `mul` instruction and one `add` instruction. It might increase kernel performance if these instructions are combined into one `mad` instruction, i.e. an instruction which performs the multiply and the addition in one instruction.

When examining a PTX kernel produced by *nvcc*, it is apparent that *nvcc* replaces many of the `add` and `mul` instructions with a single `mad` instruction. Thus, we expect that this optimization is important for the runtime performance and have decided to implement this optimization in the `Optimizer`.

The `Optimizer` introduces this optimization, by replacing a `Multiply` and a `Add` statements with a combined `MultiplyAdd` statement, if the `Multiply` statement is directly followed by the `Add` statement. Consider the list of statements: `Add`, `Multiply`, `Add`. Here, the `Optimizer` will replace the second and third statement with a combined `MultiplyAdd` statement.

This allows the `Emitter` to output a `mad` instruction as described in Section 4.5, which improves performance in situations where the kernel is compute bound and not memory bound. This replacement is possible since we know that `Declarations` are read at most once, thus the value produced by the first `Declaration`, in this case the `Multiply`, is only needed by the `Add` following the `Multiply` statement.

We expect this optimization will give speedup to all kernels which have multiplication followed by an addition, i.e. Matrix Multiplication and Black Scholes in our benchmark suite.

4.4.2 Copy Omit

Recall from Section 4.2 that the `Invoker` marshalls all required data to the device, invokes the kernel and marshalls all the data back from the device to the host. In some situations, data which is marshalled to the device is not needed, and in other situations, data which has not been modified, is marshalled from the device back to the host.

Consider the expression, $c = a + b$, if this expression was computed on the device, the a and b data must be marshalled to the device prior to execution, but c does not have to be, as it is only written to. Also, a and b do not have to be marshalled back from the device after kernel invocation is completed,since they have not been modified, but c must be marshalled since the kernel has changed c. We argue that the runtime performance will be increased by using this optimization, as we are able to eliminate unnecessary marshalling and thus reduce the runtime overhead. Therefore we decide to implement the Copy Omit optimization.

To introduce the Copy Omit optimization, the `Optimizer` traverses the list of all statements and flags the fields used by the statements depended on how they are used. In the expression above, if a, b and c were arrays, the `Optimizer` would find a `LoadElement` statement for a and one for b, and thus be flagged as being read. Similarly, the `Optimizer` would find a `StoreElement` statement for c, and flag this field as being written to. As described in Section 4.6, the `Invoker` is able to use this information to reduce the amount of unnecessary copying to and from the device.

One potential problem with this optimization is that an array used as output, might be written to on the host prior to it being used as an output array on the device. The output array is not marshalled to the device prior to kernel invocation, since the `Optimizer` deems that this array is never read on the device, thus changes made by the host are lost when copying the array back. Detecting this scenario requires that we detect where the array is used on the host, and flag the array if it is written to on the host prior to invocation of the kernel, or, that the device produces a mask which can be used to overwrite only the parts written to on device. But since we do not have any benchmarks which would encounter this problem we will not implement support for this type of scenario.

We expect this optimization will giver better performance to all benchmarks which marshalls redundant data, i.e. Vector Addition, Matrix Multiplication and Black Scholes in our benchmark suite. The factor of speedup is heavily influenced by the type of algorithm, since the overhead of marshalling data is more visible in low arithmetic intensity algorithms, e.g. we expect greater performance increase in the Vector Addition benchmark, while the optimization is expected to have little effect in the Matrix Multiplication benchmark due to its high arithmetic intensity.

4.5 Emitter

After the optimization process, the `Emitter` is run on the optimized `IR` to produce PTX code that can be loaded by the CUDA Driver API, through the `Invoker`, and executed on the device.

The `Emitter` has been implemented with the visitor pattern in mind, i.e. each `Statement` has a `Visit` method which outputs the corresponding PTX code for that `Statement`. This pattern improves the separation of concerns, which in turn improves maintainability.

4.5.1 Version and Target

PTX sources start with a `.version` and `.target` directive, which specifies the PTX version number and the desired target architecture. We use PTX version 2.2 and output code for compute capable targets of 2.0, thus the `Emitter` outputs `.version 2.2` and `.target sm_20` as this is the version and target our test setup supports, which is described in Section 5.1.

4.5.2 Types

Recall from Section 3.7, that PTX is a typed language, meaning that instructions and registers have a type associated. To facilitate this requirement when generating PTX code, each CLI type maps to a corresponding PTX type.

As an example, consider the `add` instruction in CIL, and the corresponding `add` instruction in PTX. In CIL, the type of the value produced by the `add` depends on the source types, e.g. `1 + 2` will produce a `System.Int32` value, whereas `1.0f + 2.0f` will produce a `System.Single` value. In PTX, this type information is saved along with the instruction, thus, an `add` instruction which adds two signed integers and an `add` instruction which adds two floating point values have the signatures, `add.s32` and `add.f32` respectively.

As seen, `System.Int32` is mapped to ".s32" and `System.Single` is mapped to ".f32". In addition, array types are converted to the ".u32", which is an unsigned 32bit integer and treated as a pointer to a block of memory containing the elements of the array. Recall from Section 2.1.2 that our device supports 64bit addresses, this feature is however not used since we were unable to produce PTX code which functioned correctly under 64bit addressing. But since our device does not exceed the amount of memory addressable using 32bit pointers, we do not see this as a problem. This issue is discussed in further detail in Section 6.2.4.

With regards to boolean values, the `System.Boolean` type is mapped to ".s32" where a value of zero represents false, and all other values represents true.

4.5.3 Forward Declarations

Variables must be declared prior to use and must be declared as belonging to some state space with a given type, e.g. `add.s32 a, b, c;` adds b and c together and stores the result in a. The three variables must be declared prior to use, as belonging to the `.reg` state space which is the register state as described in

Section 3.7.3, and as having the type .s32, using the following signature: ".reg .s32 variableName;".

As described in Section 4.3, each `Declaration` statement has an associated type that denotes the type of the value produced by that `Declaration`. In addition, a `Declaration` has a global name which is used by the `Emitter` to produce a variable name. For each `Declaration` statement in a `Method`, the `Emitter` forward declares a variable in the `.reg` state space which is used by other statements. Thus, all `Declaration`s saves their produced value in a register which can be used by later statements.

Local variables, which were described in Section 3.8.3, are also forward declared prior to use.

4.5.4 Branches

When the `Emitter` encounters a `Branch` statement, a `bra` PTX instruction is emitted along with a label which refers to the location to jump to. When the `Emitter` encounters the `Statement` the label belongs to, the label is outputted along with the PTX code for the particular `Statement`.

Conditional branches in our `IR` is also supported, such as the `BranchIfTrue` statement which perform the jump if some expression returns true. As mentioned in Section 3.7 PTX has guard predicates, which determines if an instruction is carried out or not based upon some guard. The `BranchIfTrue` statement is translated into the PTX code: `"setp.eq.s32 p,v,1; @p bra Label;"`, where the `setp` instruction sets the `p` variable to true, if the variable `v` is equal to 1.

The branch is taken if the guard `@p` is true, i.e. if the value of `v` is 1. The `setp` instruction can be used with other operators to implement other conditional branches such as the less than (`lt`) and greater than (`gt`) operators.

4.5.5 Arrays

Recall from Section 4.5.2 that array types are mapped to `.u32` and used as pointers into a block of memory containing the elements of the array. The first four bytes of the block contains the size of the array, i.e. how many elements are present in the array. The size is used to compute the index address of rectangular arrays, which will be covered below. The rest of the bytes contain the actual elements.

Array access statements, such as the `GetElement` and `SetElement` statements, are translated into `ld.global` or `st.global` instructions which performs a load or a store in global memory. Prior to the load or store, the address of the element is calculated using a combination of `mul` and `add` instructions. To access the element at the index i of an array with address a, four is added to a such that a points to the first element of the array, since the first four bytes contains the size of the array. Afterwards, i is multiplied with the size of an element, and added to a such that a now contains the address of the $i'th$ element in the array. Lastly, the `st.global` or `ld.global` instruction is executed which writes or reads the element at index i.

Rectangular arrays are handled in a similar manner, except that the address a is computed using two indices in the case of a two dimensional rectangular array. Array access on a two-dimensional rectangular array where each dimension has a size of: $size$, on an index: x, y and having an element size of: $elementSize$ bytes is computed using the following formula: $array[y, x] = a + 4 + (x * elementSize) + (y * size * elementSize)$.

4.5.6 Math Functions

Recall form Section 4.3.7 that some of the `System.Math` methods are translated to statements in the IR these statements are translated directly to one or more PTX instruction, e.g. the `System.Math.Sqrt` method can be translated to the `sqrt` instruction in PTX. Other methods, such as `System.Math.Exp` and `System.Math.Log`, requires a combination of PTX instructions.

4.5.7 Device Functions

The `IR` contains one or more methods that are translated to PTX. The `Emitter` traverses each `Method` and generates a device function which can be executed on the device. Each `Method` contains all information required by the `Emitter` to generate a method signature, such as parameter count, parameter types and return type of the method. In addition, the fields required by the method is also added as parameters to the method.

Consider the method:

```
static float Test(int i)
{
  return array[i];
}
```

This method takes one argument `i` of type `System.Int32`, and returns a `System.Single` value. This method makes use of one field, the `array` field. The `Emitter` translates this method into a PTX device function as shown below:

```
.func (.reg .f32 return) Test
(
  .reg .s32 P0
, .reg .u32 F0
)
{
...
}
```

The ".reg .f32 return" denotes that this function returns a value of type `f32` and that `return` is used as return register. In addition, the PTX function has two parameters, `P0` and `F0`. `P0` is the parameter of the `Test` method, and `F0` represents the field used by the device function.

A `Call` statement, which calls a method, is simply translated into a PTX function call. Registers are used to pass arguments and the return value to and from the function.

4.5.8 Kernel Function

After all device functions have been generated by the `Emitter`, a single kernel function is generated which is called by the `Invoker`. The kernel function calls the first device function generated by the `Emitter`, which is the function generated from the method body of the original `Action` delegate. In addition, the kernel function computes a linear thread index of each thread using the special registers: `tid`, `ctaid` and `ntid`, where `tid` holds the thread index relative to a CTA, `ctaid` holds the CTA index in the grid (block index in CUDA terms) and `ntid` holds the number of blocks in the grid.

In addition, the generated kernel function performs the necessary boundary checks prior to invoking the device function. As an example, a call to `Parallel.For(0,1000,body)` will spawn 1024 threads on the device, since two CTAs of 512 thread will be spawned as described in Section 4.6. The check is made to ensure that only the first 1000 threads calls the first device function.

4.6 Invoker

The invoker is in charge of initializing the CUDA Driver API, marshalling data and invoking the generated kernel on the device. The CUDA Driver API can been called by using the P/Invoke feature of CLI, this however would require that we made signatures for all the methods of the CUDA Driver API which we use. Instead, we have chosen to use the CUDA.NET library which provides these signatures for us, thus allowing us to call the CUDA Driver API [8].

Before the PTX kernel can be invoked some setup is required, e.g. the CUDA Driver API must be initialized and a CUDA context created. Next the data which is used by the kernel is marshalled to the device. Likewise, the data is marshalled back when the kernel execution has finished. As described in Section 4.5, the `PTX` object contains a reference to all fields used by the PTX kernel. This allows the `Invoker` to decide which data should be marshalled to and from the device. The kernels are invoked with a CTA size of 512, which is the maximum size allowed by compute capability 1.x devices. This size was chosen such that we are able to test the implementation using our developer computers.

When memory is marshalled back and forth, the data is pinned on the host system, using the `GCHandle.Alloc` method. This ensures that the data is not garbage collected before the data has been fully transfered. Furthermore it creates a `GCHandle` object which gives a pointer to the memory area, this is then used as input to the `cuMemcpyDtoH` or `cuMemcpyHtoD` methods, depending on the direction of the memory transfer.

Furthermore, kernel arguments must be memory aligned. This is done by adding an offset to the argument given to the `cuParam*` function, which is the function that adds the parameters to the kernel call. This offset is calculated using the alignment requirements of the host system [47, sec. 3.3.3]. For example if the current offset is 14, and the memory must be 4 byte aligned, a two byte padding must be added, such that the parameter is put at offset 16. This new offset can be calculated by the function `new_offset = offset + alignment - (offset mod alignment)` [47, sec. 3.3.3].

4.6.1 Optimizations

We have introduced three optimizations to the `Invoker`, context caching, increased L1 cache and the Copy Omit optimization.

Context Caching Before any CUDA Driver API methods can be called, a context must be created, this is done by calling the `cuCtxCreate` method. However our initial tests shows that this is a slow operation. A naive implementation of the `Invoker` will simply create a new context each time a kernel is run. Therefore, we chose to store the context object in a static field such that we only need to create the context when the first kernel is run. We expect this optimization to increase runtime performance of all benchmarks in steady state, since the context is only created once.

L1 Cache As mentioned in Section E.2, compute capability 2.x devices can use part of their shared memory as a L1 cache and the programmer has some control over how large a part this is. By default, shared memory is 48KB and the

L1 cache is 16KB [47, G.4.1] therefore we might see an increase in performance by increasing the L1 cache to its maximum size of 48KB.

Because our implementation does not utilize shared memory for any other purpose we allow CUDA to utilize a larger part of the shared memory for L1 caching than default. This optimization is introduced by calling `cuCtxGetCacheConfig` with the `CU_FUNC_CACHE_PREFER_L1` as argument. This increases the L1 cache from 16 KB to 48 KB [47, sec. G.4.1].

We expect this optimization to give speedup to all benchmarks which iterates their data several times, i.e. the Matrix Multiplication benchmark.

Copy Omit As described in Section 4.4, the `Optimizer` annotates, all fields used by the `Action`, to indicate whether the field is read from or written to. These annotations allow us to omit copying data to the device which are not read from and copy data back to the host which are not written to on the device. The `Invoker` checks these fields and only copy the data which is necessary. As described during Section 4.4 we expect this optimization will improve performance on all our benchmarks, except the Overhead benchmark which does not copy any data to or from the device.

4.7 Summary

This chapter covered the implementation of APL and its four components, the `Parser`, which parses the CIL and generates `IR`, the `Optimizer`, which performs optimizations on the `IR`, the `Emitter`, which traverse the `IR` thereby generating PTX code, and the `Invoker`, which invokes the PTX kernel on the device.

All functional requirements described in Section 4.1.1 have been implemented, which means that all of our benchmarks can be executed. In addition to the functional requirements, optimizations have been introduced to increase the efficiency and scalability of the solution, which are both important non-functional requirements as described in Section 4.1.2.

Five optimizations were introduced: The PTX caching optimization was introduced into the `Driver` which caches the produced PTX kernel. We expect that this optimization will increase runtime performance after the initial compilation of the PTX kernel.

The fused multiply add optimization was introduced into the `Optimizer`, which is an optimization that is also used by *nvcc*. We expect that this optimization will give a speedup in the Matrix Multiplication and Black Scholes benchmark, as it reduces two instructions to one.

The Copy Omit optimization was introduced in the `Optimizer` and `Invoker`, which reduces unnecessary copies to and from the device. We argue that this optimization will provide a speedup, since marshalling of data to and from the device is an expensive process.

Lastly, context caching and L1 caching was introduced into the `Invoker`. Context caching should remove the overhead of creating a context in subsequent calls to APL, thus increasing the runtime performance in steady state.

Increasing the L1 cache should increase performance in benchmarks which performs multiple access to the same memory locations, such as is the case with the Matrix Multiplication benchmark.

These optimizations are benchmarked in Section 5.2, and their results are analyzed.

5 Results

The aim of this chapter is to cover the results of running our benchmark suite, containing the Overhead, Vector Addition, Matrix Multiplication and Black Scholes benchmarks, which were all described in Appendix C.

Recall that we have four implementations of the benchmark suite, namely the C# implementation which uses sequential for-loops, the C# implementation which uses the TPL, the C# implementation which uses APL, and the CUDA C implementation. The results of running these benchmarks will be covered in this chapter.

The hardware configuration and software configuration of the test setup will be covered in Section 5.1. Afterwards, to determine if our optimizations have any effect, the results of benchmarking APL using different combinations of optimizations is covered in Section 5.2.

Lastly, Section 5.3 and Section 5.4 compares the performance results of APL with the performance results of the three other implementations, where the first section compares the difference in performance and the last section covers the difference in scaling with regards to input size.

The array sizes of the benchmarks in Section 5.2 and Section 5.3 have been fixed to $33,000,000$ floating point elements for the Vector Addition and Black Scholes benchmarks, and $768x768 = 589,824$ elements for the Matrix Multiplication benchmark. The Overhead benchmark requires no data, but it performs $33,000,000$ iterations.

By running the benchmark implementations we saw that all implemented benchmarks computed the correct values. The benchmarks runner is in charge of validation, which is done by comparing the values computed by each benchmark implementation with all other computed values of that benchmark, as described in Appendix D.

5.1 Test Setup

In this section, we will present the hardware and software setup used to perform the benchmarks.

CPU	Intel Xeon E5420: Quad Core, 2.5 GHz 12MB cache
RAM	2 X 2GB Samsung PC2-5300
GPU	Zotac Nvidia GeForce GTX 470: 1280MB GDDR5 RAM, 16X Peripheral Component Interconnect Express (PCIe) 1.0
Hard Drive	Seagate Barracuda 7200.12: 500GB, 7200rpm, 16MB buffer
Chipset	North Bridge: Intel 5400A, South Bridge: 6321ESB

Table 5.1: LENOVO ThinkStation D10 6427H6G

OS	Microsoft Windows 7 Enterprise x64
Graphics Driver	Nvidia GeForce 270.21 WHQL
BIOS	2XKT31AUS
CLR	v4.0.30319
.NET Framework	4.0

Table 5.2: Software used

5.1.1 Hardware

With regards to the hardware used, we have access to the same ThinkStation as we used on the 9th semester. This time, however, it is fitted with a newer GPU. The hardware specs are shown in Table 5.1.

5.1.2 Software

With regards to the software used, we use the same operating system as we did at our 9th semester, but the graphics driver has change. The software specifications are shown in Table 5.2.

5.2 Optimizations

The aim of this section is to benchmark the five optimizations which were introduced in Chapter 4. The five optimizations are:

1. More L1 cache
2. CUDA Context caching
3. PTX caching
4. Fused multiply add
5. Copy Omit

Each optimization will be benchmarked separately to determine if they have an effect on performance. Lastly, all optimizations are turned on and their performance is compared with a benchmark run where all optimizations have been turned off. The result of benchmarking these optimizations can be seen in the following figures, Overhead benchmark: Figure 5.1, Vector Addition: Figure 5.2, Matrix Multiplication: Figure 5.3 and Black Scholes: Figure 5.4.

In the following sections we discuss the result of each optimization, along with how the optimizations perform when combined.

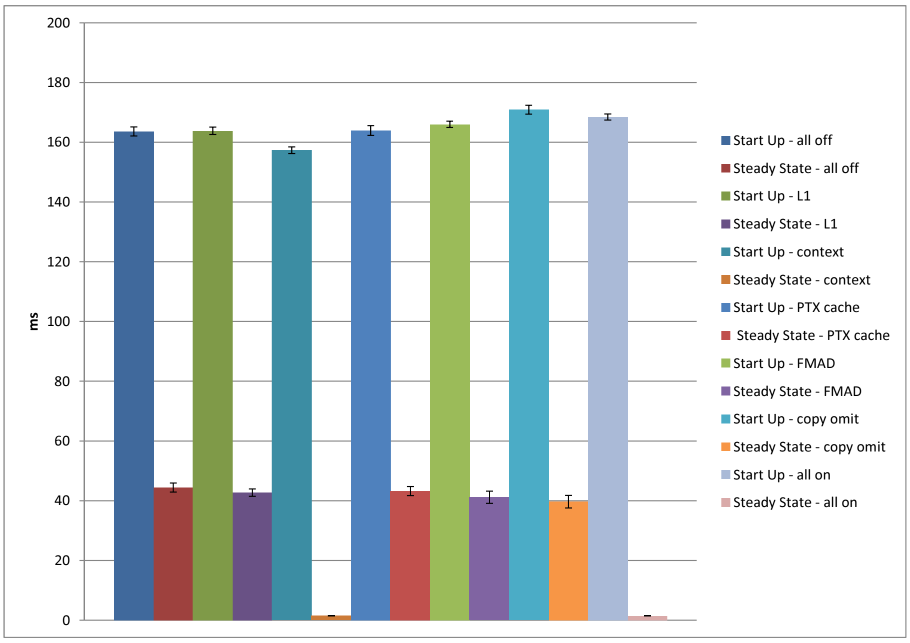


Figure 5.1: Result of the different optimizations for the Overhead benchmark.

5.2.1 More L1 Cache

This optimization works by increasing the memory area used for L1 caching from 18 KB to 48 KB. This optimization will affect computations which accesses the same memory several times.

Matrix Multiplication is the only benchmark which iterates over the same memory multiple times. This is also reflected in our results, in that all benchmarks, except Matrix Multiplication, gives overlapping confidence intervals with and without the L1 cache optimization. While Matrix Multiplication gains a speedup of 1.12x at startup and 1.16x in steady state as seen on Figure 5.3. This result match our expectation stated in Section 4.6.1.

5.2.2 CUDA Context Caching

Caching the CUDA context decreases the runtime of all benchmarks in steady state by 36-44 ms. The startup time is also affected a little by this optimization as shown by the Overhead and Vector Addition benchmarks where the confidence intervals do not overlap. This increase in performance is however very small and we expect that the difference might be caused by the context not having to be destroyed for each run. In steady state a greater speed up is seen in all benchmarks, since the context for all consecutive calls then have been cached. This result also matches the expectations described in Section 4.6.1.

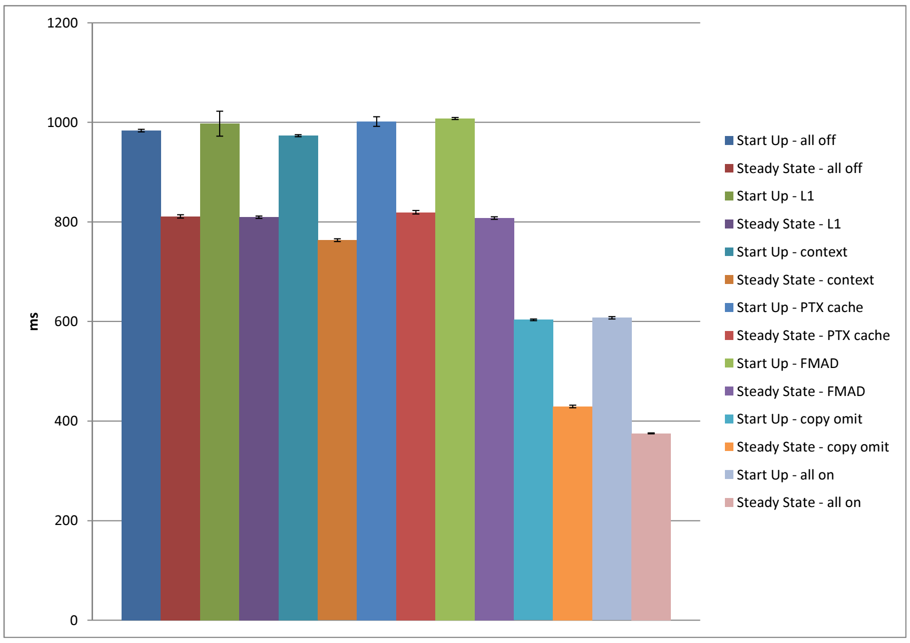


Figure 5.2: Result of the different optimizations for the Vector Addition benchmark.

5.2.3 PTX Caching

By caching the generated PTX code of an `Action`, we avoid having to recompiling the same `Action` for consecutive calls. The benchmark, however, only shows non-overlapping confidence intervals in the Vector Addition benchmark, when comparing PTX caching enabled with PTX caching disabled and in this case it is a slowdown. This result is not as our expectations described in Section 4.2. We expected that this optimization would result in speedup for all benchmarks in steady state.

The reason for this is likely that the overhead of compiling the code to PTX is not big enough to be decreased by using a cached PTX. If the method of the `Action` was greater in size, or the `Optimizer` performed more transformations, thus making the compilation take longer, this optimization might provide increased performance.

With regards to the Vector Addition benchmark, a 1.01x slowdown is observed, which can be due to the Vector Addition kernel is very small, only a few statements, thus PTX caching introduces too much overhead.

We have experimented with The Native Image Generator tool [31], which allow pre-JITing of .NET code prior to execution. Here, we see that the Vector Addition benchmark does not perform worse when PTX caching is enabled. This indicates that the overhead of JITing the .NET code might be the reason for the slowdown.

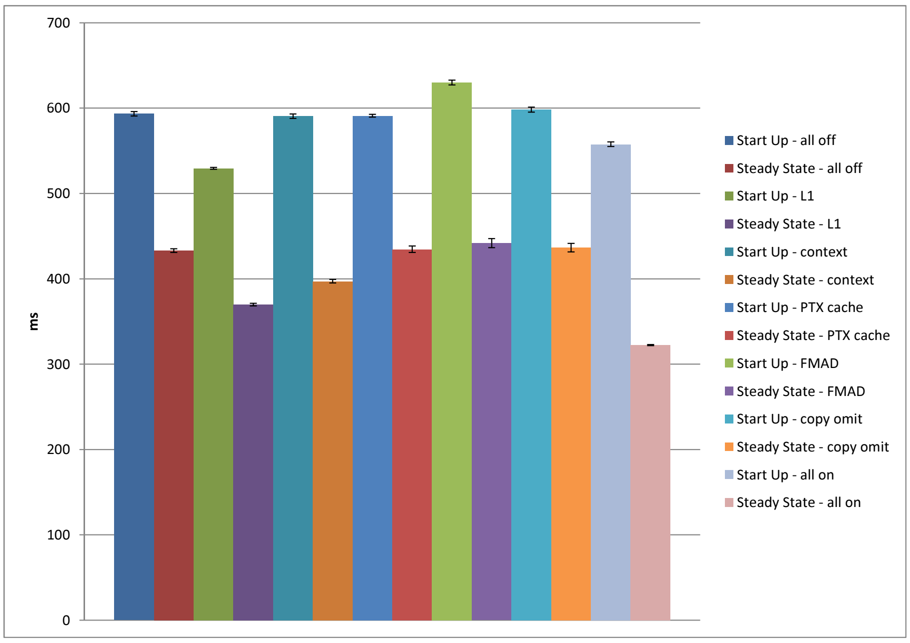


Figure 5.3: Result of the different optimizations for the Matrix Multiplication benchmark.

5.2.4 Fused Multiply Add

By using the `fmad` instruction, we can replace two instructions, i.e. the `mul` and `add` instruction, with a single instruction which should provide a speedup.

Our results show however that the `fmad` have no effect in most benchmarks. The only benchmark which do not produce overlapping confidence intervals both in steady state and startup is Matrix Multiplication which shows a slowdown, 54.5 ms on startup and 24 ms for steady state, i.e. 1.09x and 1.05x slowdown. This can be explained by Matrix Multiplication being the benchmark which has the most instructions merged into `fmad` instructions.

The reason for the slowdown when applying `fmad` can be explained by the time spend traversing the IR instructions and inserting the `MultiplyAdd` instruction overshadows the potential speedup. The reason that steady state is faster than startup can be that the code which traverse the IR code only has to be JIT compiled by the CLR the first time it is run.

Our expectations described in Section 4.4 were that this optimization would lead to speedup for the benchmarks which used consecutive multiplications and additions, i.e. Matrix Multiplication and Black Scholes, the results however did not meet our expectations.

5.2.5 Copy Omit

In order for the copy omit optimization to show speedup, a large amount of memory copies must be omitted compared to the arithmetic intensity of the kernel, e.g. Matrix Multiplication which operates on $3x589,824$ float values, is

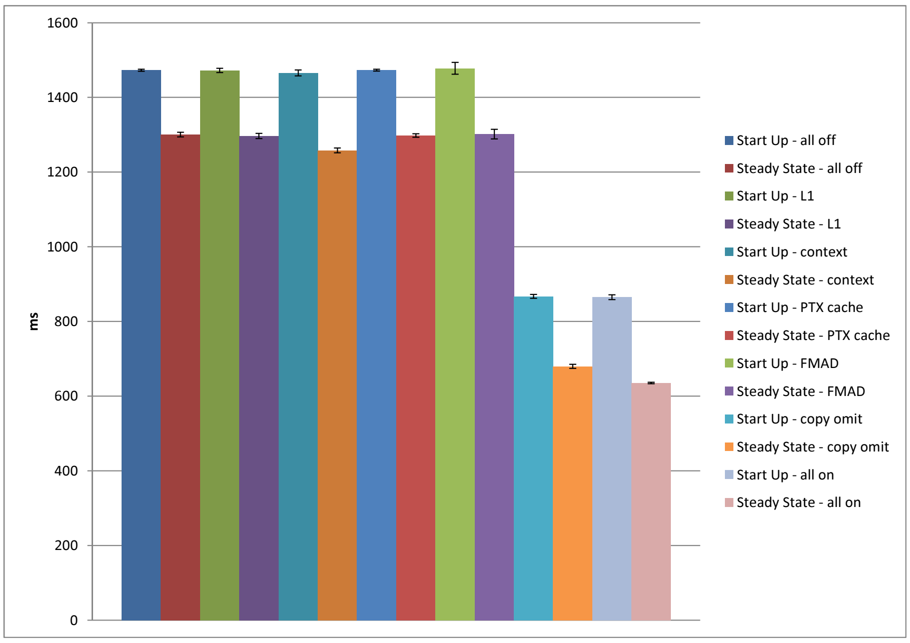


Figure 5.4: Result of the different optimizations for the Black Scholes benchmark.

very arithmetic intensive, and produces overlapping confidence intervals. Vector Addition, which is not arithmetic intensive, achieves a speedup of 1.61x for startup and 1.89x in steady state, and operate on a input of $3x33,000,000$ floats which is significantly larger than the data used by the Matrix Multiplication benchmark. This trend is also reflected by the Black Scholes which operate on $5x33,000,000$.

Our expectations described in Section 4.4 stated that we expected to achieve speedup for all benchmark marshalling redundant data, i.e. Vector Addition, Matrix Multiplication and Black Scholes. This is not fully coherent with the results described above, since Matrix Multiplication did not achieve a speedup, but this can be explained by the arithmetic intensity of the kernel. We also stated that we expected the speedup factor to be greater for benchmarks with smaller runtime, as the time spend copying data would be a larger part of the overall time. This was found to be true, e.g. Vector Addition and Black Scholes sees speedups very close to each other with a speedup of 1.89x and 1.91x respectively, even though the overall time of Black Scholes is 489 ms longer. This may seem to be against our expectation, but Black Scholes avoids five copies and Vector Addition only three, thus giving additional speedup to Black Scholes resulting in the speedup factor.

5.2.6 All Optimizations On

When looking at the results for the benchmarks with all optimizations enabled, we see that for the Overhead, Vector Addition and Matrix Multiplication benchmark, it the case that in startup there is at least one of the other configurations which is faster, e.g. for Matrix Multiplication Figure 5.3 it is faster when only L1 cache is enabled in startup compared to when all optimizations are enabled. However, in steady state, the benchmark with all optimizations enabled is always the fastest.

5.2.7 Summary

We have seen that the results very from optimization to optimization and from benchmark to benchmark. Some optimizations results in a slowdown, while some optimizations results in a speedup. A general trend and important factor when developing optimization is that the optimization have to give a large enough speedup to not be absorbed by the overhead imposed by making the optimization, especially if it is an `Action` which is only run once. Furthermore, the time spend by the CLR on JIT compiling the code performing the optimization may be an important factor, though it only imposes an overhead the first time the code is run.

In particular we see that Copy Omit for large inputs gives a high speedup. L1 caching gives better performance as well for benchmarks which iterate the input several times. Context Caching sees a speedup for all subsequent calls to APL. PTX caching and Fused Multiply Add resulted in no particular speedup, and in some cases even slowdown.

With regards to our expectations, we saw that More L1 Cache, Context Caching and Copy Omit behaved as expected. PTX Caching and Fused Multiply Add did not meet our expectations and in some cases even resulted in worse performance.

5.3 Comparisons

In this section, we will present and reflect on the result from each of the individual benchmark suite, i.e. the CUDA C, TPL, APL and sequential for-loop benchmark suites.

The results of each benchmark will be compared in the following sections.

5.3.1 Overhead

The startup time of APL is high compared to all other implementations, especially the sequential for-loop has a low startup runtime, as seen in Figure 5.5. The reason the sequentially executed loop is faster, may be that it has minimal overhead, whereas all other implementations have higher overhead, e.g. creating a CUDA context which we saw in Section 5.2 is a slow process.

The steady state for APL however, is 9.3x faster than the for-loop and 82.5x faster than the TPL implementation. The only implementation which exceeds APL in speed in steady state is the CUDA C implementation, with less than a millisecond.

5.3.2 Vector Addition

Overall, the APL implementation of Vector Addition is slower than the other implementations, except for the CUDA C implementation in steady state, as

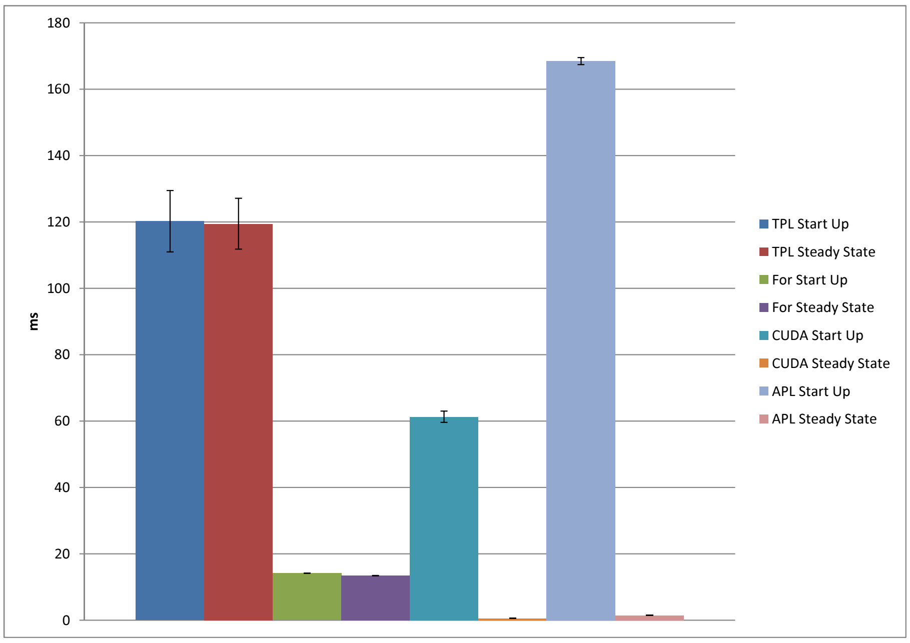


Figure 5.5: Result of benchmarking the different implementations of the Overhead benchmark.

seen in Figure 5.6. The worst case slowdown is the sequential for-loop, which is 2.6x faster than the APL implementation. One reason that APL is slower than the implementations which run on the CPU could be that only one operation is made on each element in the array. Thus the time spend marshalling data to and from the device exceeds the speedup from executing on the GPU. As seen, the sequential for-loop even beats TPL in speed. This might be due to the overhead of spawning CPU threads, which is not amortized by the low arithmetic intensity of the vector addition, or that false sharing is observed.

5.3.3 Matrix Multiplication

For Matrix Multiplication the CUDA C implementation, gives the best performance with a speed up of 24.7x compared to APL. The main reason for this is most likely that the CUDA C implementation is more advanced, e.g. memory is split in smaller chunks and stored in shared memory, to avoid access to high latency global memory. The APL implementation is only able to use shared memory as a L1 cache.

Comparing APL with TPL we see that the startup for APL gives a speed up of 1.5x, and a speed up of 2.58x in steady state. See Figure 5.7. A greater speedup is achieved when comparing against the sequential for-loop, 5.8x for startup and 10x for steady state. The reason that we see this speedup for Matrix Multiplication and not Vector Addition is that Matrix Multiplication is a more arithmetic intensive algorithm than Vector Addition.

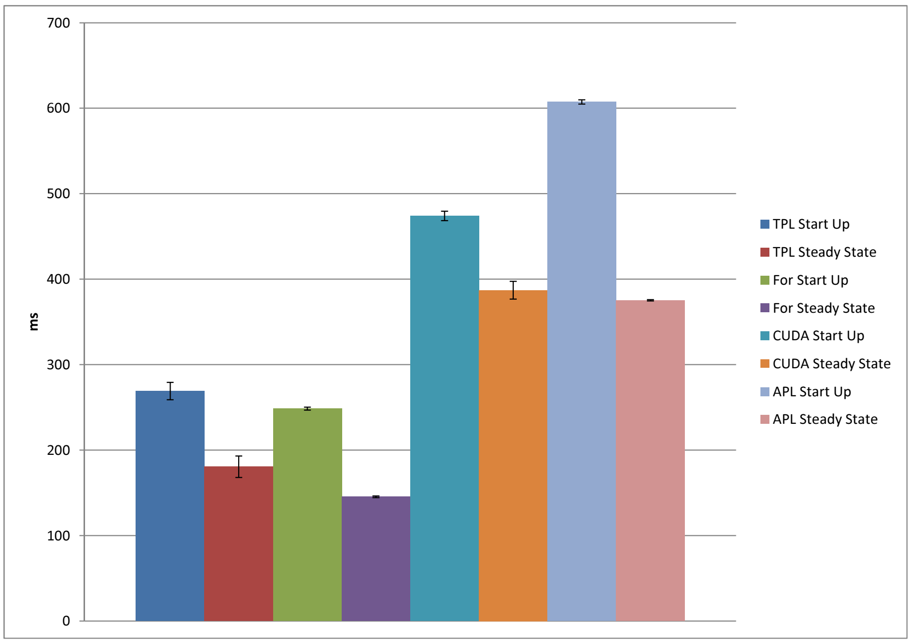


Figure 5.6: Result of benchmarking the different implementations of the Vector Addition benchmark.

5.3.4 Black Scholes

APL is faster than all other implementations, except for CUDA C at startup as seen on Figure 5.8. In particular, the sequential for-loop is slow compared with APL, which is 14x faster for steady state and 10x faster at startup. The reason that APL startup is slower than CUDA C, but APL steady state is not, may be the overhead of the CLR JIT compiling the APL code, since this will only be experienced during the first run. The best speedup experienced is 14x, when comparing APL with the sequential for-loop in steady state. APL is 1.02x faster than CUDA C in steady state while CUDA C is 1.2x faster than APL in startup.

5.3.5 Kernel Time

In Section 4.1.2, we stated that in order to fulfill efficiency, APL must generate PTX kernels which performs as well as their CUDA C counterparts. Figure 5.9 shows the kernel times for APL and CUDA C in startup and steady state for each benchmark.

The Overhead benchmark shows around 0,8 ms difference between the CUDA C kernel time and APL kernel time. This may be because the APL kernel for each thread calculate its thread index based on the thread block and grid block as described in Section 4.5.

The kernel time for Vector Addition in CUDA C and APL are very close, with a difference of 0,12 ms. We argue this is because of the similarity of the implementations, e.g. none of the implementations utilize optimizations which

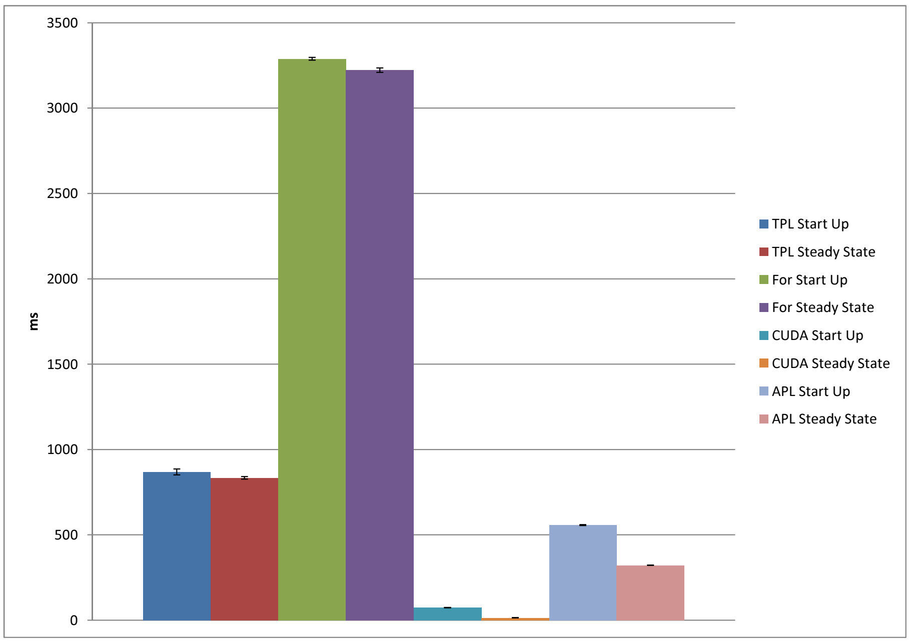


Figure 5.7: Result of benchmarking the different implementations of the Matrix Multiplication benchmark.

affect the result that the other does not use. The small difference that we do see might be caused by the APL implementation calling a device function witch the CUDA C implementations dose not.

For Matrix Multiplication CUDA C is 64.95x times faster than the APL implementation. This large time difference might be caused by the extensive optimizations implemented in the CUDA C version, e.g. the memory is split into smaller blocks and stored in shared memory before being operated on. For more on the CUDA C optimizations see [47, sec 3.2.2].

The Black Scholes kernel is around twice as fast in the CUDA C implementation as the APL implementation. This may be due to *nvcc* better optimizes arithmetic instructions and does less loads from global memory, APL does 11 global loads where the *nvcc* generated PTX code only performs 3 global loads.

5.3.6 Summary

APL is in general faster than the sequential for-loop and TPL implementations. The sequential for-loop and TPL implementations are however faster than both the APL and CUDA C implementations of Vector Addition. The CUDA C Matrix Multiplication implementation achieves very good results, 24.7x times faster than APL, this is because it is heavily optimized to take advantages of shared memory. APL Black Scholes is only beaten by CUDA C in startup, but beats CUDA C in steady state.

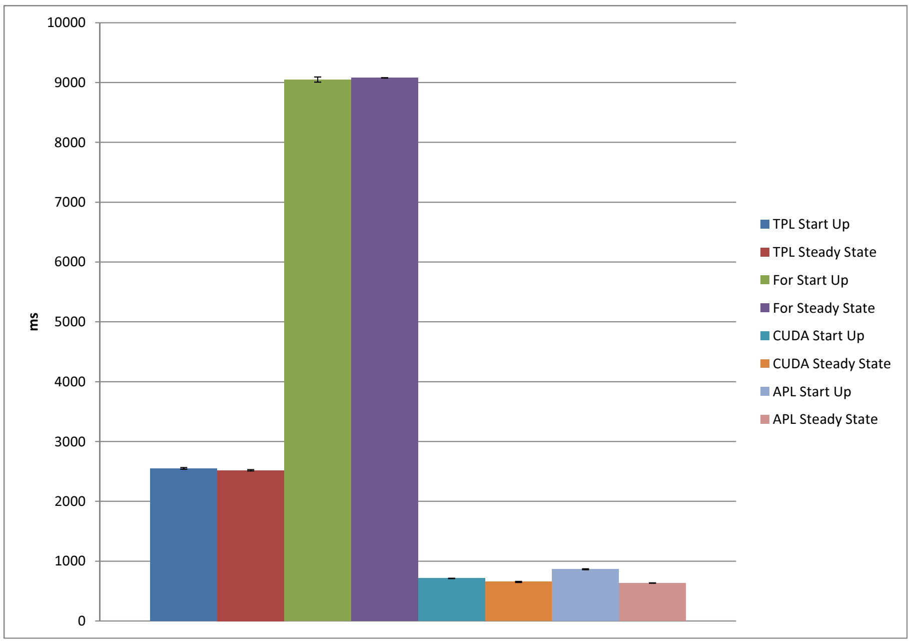


Figure 5.8: Result of benchmarking the different implementations of the Black Scholes benchmark.

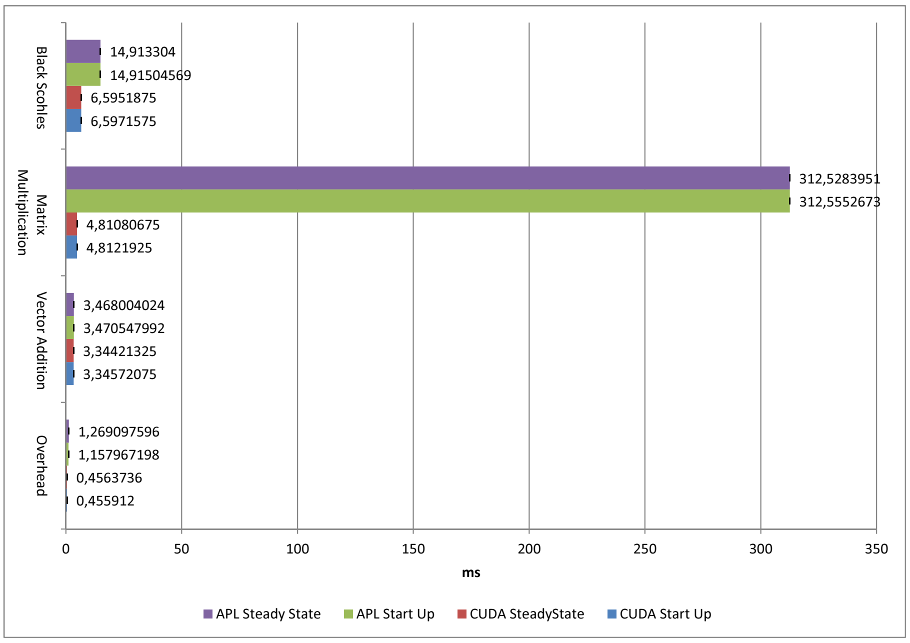


Figure 5.9: Shows the time used executing the kernel for each benchmark in startup and steady state.

5.4 Scaling

Recall from Section 4.1.2 that scalability is an important non-functional requirement. The aim of this section is to cover how well our APL benchmarks scale with regards to the input size, compared to the other implementations.

The Matrix Multiplication benchmark found in the CUDA SDK only supports matrices where each dimension has a size which is a multiply of 16. We have to take this into account when choosing the input sizes used to investigate how well it scales, since choosing a size which is not a multiply of 16 will break the CUDA C implementation of the benchmark. Also, the GeForce 470 GTX is limited to running at most 33553920 iterations in the `Parallel.For` loop, since the `Invoker` uses a CTA size of 512 as stated in Section 4.6, and a kernel can at most be invoked with 65535 CTAs [47, sec. G.1].

Based on these restrictions on input size, we have chosen to investigate the scalability of APL with all optimizations enabled by running the benchmark suite with different input size starting with a size corresponding to a 16×16 matrix and doubling the size in each dimension for each run until the limit on the number of iterations is reached. This approach gives a sequence of input sizes which is: 256, 1024, 4096, 16384, 65536, 262144, 1048576, 4194304 and 16777216, thereby quadrupling the input size every iteration.

5.4.1 Overhead

Figure 5.10 shows that APL and CUDA C scales almost identically in the overhead benchmark, despite APL always being a little slower. We also see that for small inputs, both CPU implementations perform better than APL, but as the input size increases, APL begins to perform better than the CPU implementations. One reason that the sequential for-loop appears to scale worse then APL, might be that the loop condition must be checked for each iteration of the loop while APL just starts a kernel with the given number of threads and does the boundary checking in parallel. One reason for TPLs poor scaling might be that TPL tries to balance the work between the different CPU cores, and since no work is performed in the loop, the overhead introduced by the load balancing overshadows any speedup introduced by parallel execution.

5.4.2 Vector Addition

Figure 5.11 shows that APL and CUDA C scale almost identically in the Vector Addition benchmark, but that CUDA C is a little faster than APL. All of the implementations appear to scale similarly, but the implementation which uses the sequential for-loop is much faster than the other implementations, regardless of the input size. This is most likely because the sequential for-loop does not have the overhead of marshalling data back and forth to the device, as is the case with the APL and CUDA C implementations, and does not have the risk of scheduling overhead and false sharing, which might be the case with TPL.

5.4.3 Matrix Multiplication

Looking at Figure 5.12, it appears that CUDA C scales better than APL and is generally faster. This difference of scaling and performance is most likely due to the matrix multiplication algorithm used by CUDA C, in order to make efficient used of shared memory. APL does however appear to scale much like the CPU implementations until the run with 4194304 elements. At this point, the CPU implementations decrease in performance. This decrease is most likely caused by the size of the matrices, which consumes 48MB of memory in total, thus exceeding the 12MB cache of the CPU.

5.4.4 Black Scholes

Figure 5.10 shows that APL and CUDA C scale almost identically in the Black Scholes benchmark. APL is always a little slower than CUDA C. In steady state and for small inputs, the CPU implementations are faster than the GPU implementations. As the input size increases, the GPU implementations become faster than both CPU implementations.

5.4.5 Summary

APL and CUDA C scale equally well in all benchmarks, except the Matrix Multiplication benchmark, but APL is a little slower than CUDA C. The reason that the CUDA C and APL Matrix Multiplication implementations do not scale equally well, is that the CUDA C implementation is highly optimized to use shared memory.

The CPU performs better on small inputs in most of the cases, but is outperformed by the GPU for larger inputs. One exception is the Vector Addition benchmark, where the CPU is always faster due to the low arithmetic intensity of this benchmark.

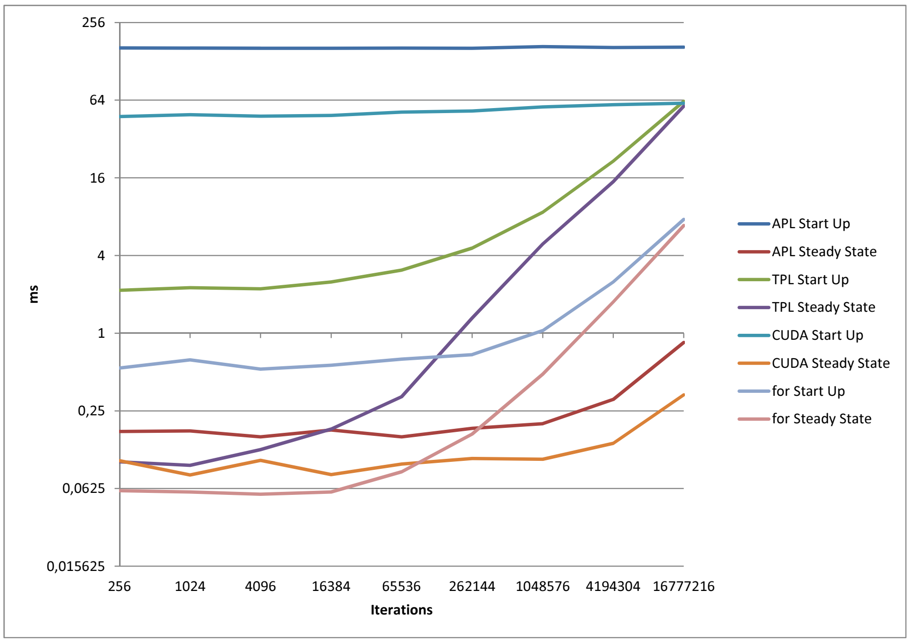


Figure 5.10: Graph showing how the implementations of the Overhead benchmark scales.

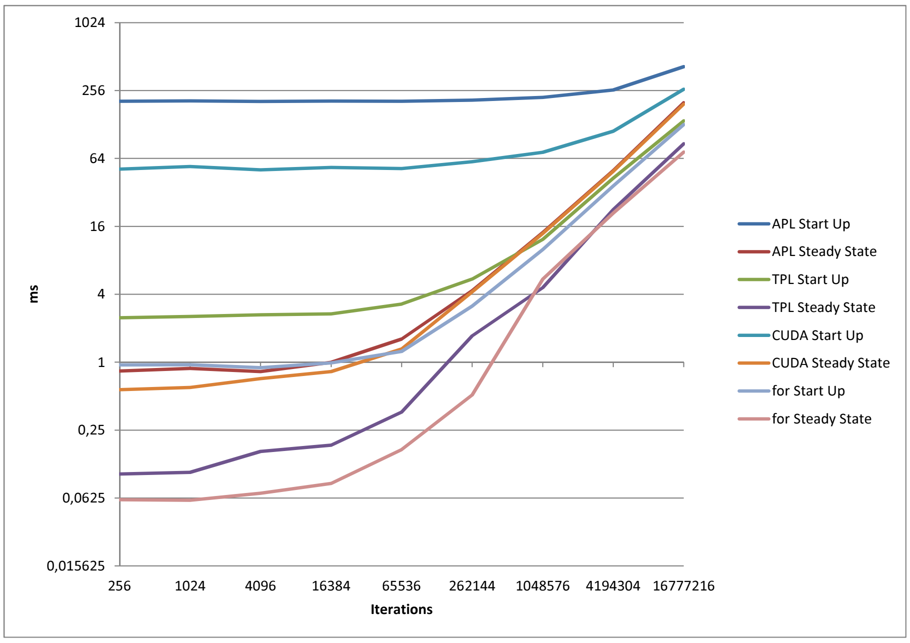


Figure 5.11: Graph showing how the different implementations of the Vector Addition benchmark scale.

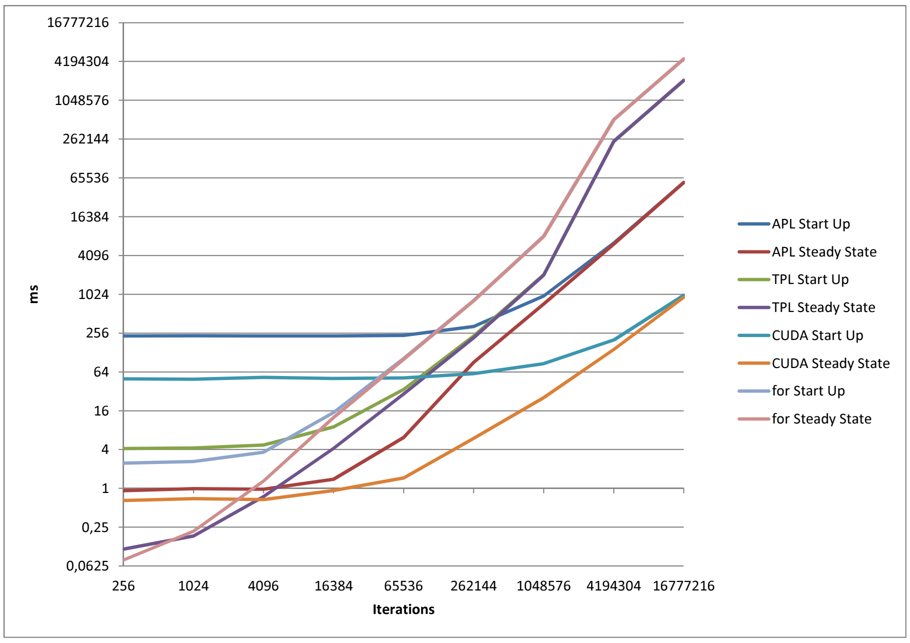


Figure 5.12: Graph showing how the different implementations of the Matrix Multiplication benchmark scale.

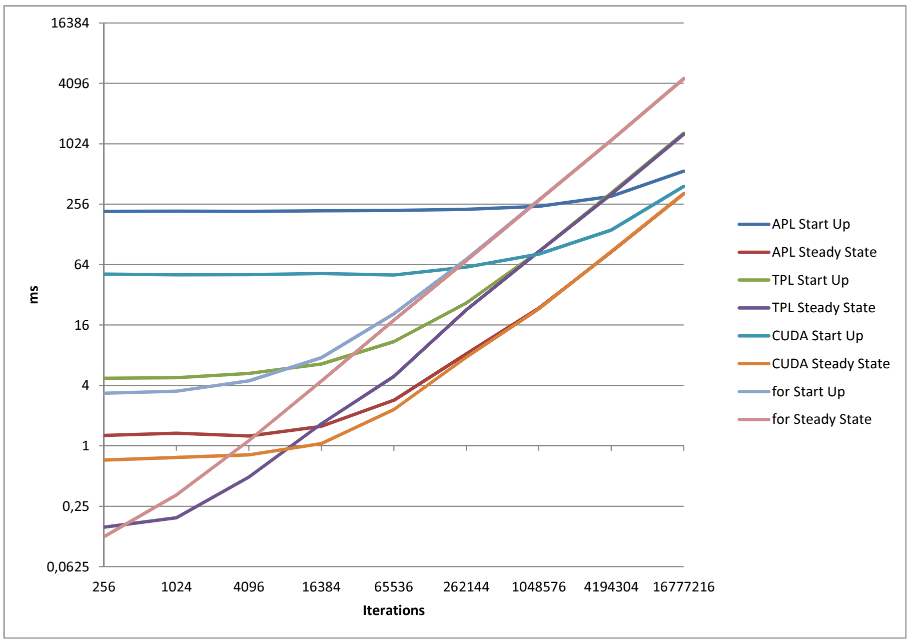


Figure 5.13: Graph showing how the different implementations of the Black Scholes benchmark scale.

5.5 Summary

In Section 5.2, we found that three out of five optimizations gave a performance increase. The PTX caching and Fused Multiply Add optimizations gave no speedup, which was not as expected. Increasing the L1 cache gave speedup in the Matrix Multiplication benchmark, as was expected due to Matrix Multiplication iterates over the same memory multiple times. Context caching gave speedup for all subsequent calls to APL, since context creation is a relatively expensive process. Copy Omit was the optimization which provided best speedup, which was as expected, since marshalling of memory to and from the device over the PCIe bus takes a long time.

In Section 5.3, we saw that the APL benchmark implementations are in general faster than both sequential for-loop and TPL. Sequential-for and TPL are both faster than APL and CUDA C in the Vector Addition benchmark. With regards to CUDA C, we see that the CUDA C Matrix Multiplication implementation is 24.7x times faster than APL's implementation due to its heavy use of shared memory. The CUDA C implementation of Black Scholes was however beaten by the APL implementation in steady.

In Section 5.4, we saw that APL and CUDA C scales equally well in all benchmarks, except for the Matrix Multiplication benchmark. This was because CUDA C implementation was optimized with shared memory, while the APL implementation only uses shared memory as L1 cache. We also saw that the CPU implementations performed better on small inputs, while the GPU performed better than the CPU implementations on larger input. The only exception being the Vector Addition benchmark, which always performed better on the CPU due to the low aritmetic intensity of this benchmark.

6 Epilogue

In this chapter, we conclude and reflect upon the achieved results and the project as a whole.

The conclusion is covered in Section 6.1, where we conclude on the project as a whole, conclude upon our results, and revisit our problem formulation thereby concluding upon our main problem and answering our sub-questions from Section 2.3.

Afterwards in Section 6.2, we discuss the results along with our choice of development method, choice of technologies, and among other things, discuss the usefulness of our solution.

Finally, we look at the future development of Accelerated Parallel Library (APL) in Section 6.3, where we discuss the potential optimizations that can be applied to the existing solution, along with new features that we would like to see added.

6.1 Conclusion

As we saw during our 9th semester project [16], solving computational problems using the Graphics Processing Unit (GPU) can provide a substantial speedup, but requires that the programmer is proficient with one of the General-Purpose computations on Graphics Processing Units (GPGPU) languages, such as Compute Unified Device Architecture (CUDA) C and its executional model. This proficiency requirement is also seen with multi-core programming on the Central Processing Unit (CPU), especially with lock-based programming, but has been alleviated to some extent by the introduction of abstractions such as the Task Parallel Library (TPL), which takes care of many low level details such as spawning threads.

In this project, we have designed and implemented APL which can execute a delegate method on the GPU using an Application Programming Interface (API) similar to that of TPL. TPL is library which can execute a delegate method on the CPU. Specifically, we have implemented the `Parallel.For` method and thereby provide the same abstractions as is provided by TPL, but using the GPU as the execution unit instead of the CPU. This has a major advantage for programmers already using TPL, since they can, with minimal change to their existing code base, move their computations from the CPU to the GPU and in some cases achieve a speedup.

In Chapter 3, it was decided that APL should utilize Common Intermediate Language (CIL) as the source language, and Parallel Thread Execution (PTX) as target language. In addition, we decided in Section 3.8 based upon the limitation of Common Language Infrastructure (CLI) expression trees, to implement our own CIL parser and use a Static Single Assignment (SSA) based Intermediate Representation (IR) to help us translate between the stack-based CIL to the register based PTX.

To guide the development of APL and benchmark the potential speedup of moving computations from the CPU to the GPU, four benchmarks were devised, ranging from simple benchmarks to more complex as described in Appendix C. Each benchmark provided concrete requirements to APL, thereby providing us with functional requirements, in addition to the non-functional requirements that we devised.

The Overhead benchmark served as the most basic case, where the only requirement was that the APL compiler must be able to produce an empty but valid PTX kernel which could be executed on the GPU. The Vector Addition benchmark introduced the requirement that we must be able to support single dimensional arrays, marshaling to and from the GPU, and arithmetic addition. The Matrix Multiplication benchmark introduced the requirement that we must be able to support arithmetic multiplication, rectangular arrays, and branching. Lastly, the Black Scholes benchmark introduced the requirement of function calls and mathematical functions.

In addition to these requirements, five optimizations were introduced in Chapter 4: The PTX caching optimization, the Fused Multiply Add optimization, the Copy Omit optimization, the Context caching optimization and the increased L1 cache optimization.

Using our benchmarks, we found that the Context caching optimization, the Copy Omit optimization and L1 cache optimization did increase performance in some of the benchmarks. Fused Multiply Add and PTX caching did however not provide any measurable speedup, at least for this set of benchmarks, and in some cases resulted in slowdown.

To determine the usefulness of APL, i.e. if a speedup could be gained by using APL instead of TPL or CUDA C, the four benchmarks were also implemented in CUDA C, C# using sequential for-loops and C# using TPL.

The Overhead benchmark showed that APL and CUDA C were superior to both TPL and sequential for-loops with a speedup of 82.5x compared to TPL and a speedup of 9.3x compared to the sequential for-loop.

The Vector Addition benchmark showed that APL was a few milliseconds faster than the CUDA C implementation, but achieved less than half of the performance of TPL and the sequential for-loop.

The Matrix Multiplication benchmark showed that the hand optimized CUDA C version was 24.7x times faster than APL. APL did however achieved a better performance than TPL and sequential for-loops with a speedup of 2.57x and 10x respectively.

The Black Scholes benchmark showed that APL is generally the fastest of all implementations. APL is faster than CUDA C by a few milliseconds, but achieves 3.96x speedup compared to TPL and 14,29x speedup compared to the implementation that uses a sequential for-loop.

6.1.1 Problem Formulation Revisited

At the beginning of this project, we created a problem formulation as described in Section 2.3, which describes our main problem and a number of sub-questions, which should help us solve the main problem. Each of the sub-questions will be addressed below and the main problem at the end of this section.

How can source code be accessed at runtime, compile-time or in any other way? In Section 3.3 we found that we had two possibilities with regards to source language: A high-level languages, such as C#, or the low-level language CIL. By using a high-level language, we would restrict the solution to just one source language.

But since we chose CIL, we are able to support all high-level languages which are compiled to CIL and at runtime access the resulting CIL code by using reflection.

How is device code best generated from .NET? The compilers available for CUDA C, Open Computing Language (OpenCL) C and PTX were analyzed in Section 3.4 and we found that OpenCL was able to compile its high-level code, OpenCL C, at runtime, and Nvidia's *nvcc* compiler was able to Ahead-Of-Time (AOT) compile CUDA C code. We also found that the CUDA Driver API supports loading of low-level PTX code, which is Just-In-Time (JIT) compiled and optimized to run on the device.

We found that PTX was the best choice since it supports more features, e.g. recursion, which are not supported in OpenCL C, furthermore we avoid transforming the low-level CIL code into a high-level abstraction, such as CUDA C or OpenCL C, and are therefore able to make a near one-to-one translation of CIL to PTX.

Which optimizations are important when generating code? Throughout Chapter 4, we implemented several optimizations: Context caching, larger L1 cache, PTX caching, Copy omit and Fused multiply add. In Section 5.2, we analyzed the performance improvements of these optimizations and saw some speedup, especially in steady state.

Some of the optimizations did however not give any speedup, and some gave slowdown. Slowdown was often seen for startup benchmarks, that is when the benchmark was run for the first time. We argue that this slowdown is caused by the Common Language Runtime (CLR) JIT compiler, which is run once on the CIL code [39]. The most important optimizations was the copy omit and context caching, since these optimization provided good speedup for several of the benchmarks. Thus, optimizations which reduces memory marshalling and memory bandwidth usage are important.

How does device code generated by our solution fare against hand written device code? To analyze the performance of our library against the performance of CUDA C, we ran several benchmarks. These results show that CUDA C in general is faster than APL, with two exceptions. The APL implementation of Vector Addition and the Black Scholes benchmark are in steady state faster than the CUDA C implementations. Vector Addition is

1.03x faster and Black Scholes is 1.02x faster. CUDA C's kernel was however 64x times faster than APL in the Matrix Multiplication benchmark, due to its heavy use of shared memory.

How does APL fare against the TPL? APLs performance is better for all benchmarks, with the exception of the Overhead benchmark in startup, and the Vector Addition in both startup and steady state. We argue that the reason Vector Addition is slower is that the number of computations is too small, because only one computation is performed per element, the performance gain is overshadowed by the time spend compiling the code and marshalling data. Vector Addition is up to 2.5x faster on TPL. APL is 82x faster in the Overhead benchmark in steady state, 2.58x faster in the Matrix Multiplication benchmark in steady state and 3.9x faster in the Black Scholes benchmark in steady state. Thus, in three out of four cases, APL is faster than TPL.

Main Problem The main problem was: "Is it possible to create a library similar to the Task Parallel Library from .NET 4.0, where the data parallel operations are executed on a many-core GPU instead of a multi-core CPU?".

Since we have implemented a library which has the same interface as TPL and is able run the same code for the benchmarks we chose to implement and achieve speedup in some cases, we rate our main problem solved.

6.1.2 Requirements Revisited

Non-Functional Requirements

In Section 4.1.2, we decided upon a number of non-functional requirements, which were rated based upon how much focus we should put into the particular non-functional requirement. In the following we will conclude upon how well we fulfilled the very-important, important and less important non-functional requirements.

Efficiency - Very Important - Partially fulfilled *The economical utilization of the technical platform's facilities*
In order for this requirement to be fulfilled, we stated that the performance of the produced PTX kernel from APL should be the same as the CUDA C kernel. Furthermore, the overhead of APL should be comparable to that of the CUDA C implementation.

In Section 5.3.5 we saw that the kernel runtime of benchmarks implemented in CUDA C always has better performance than the APL implementations kernel runtime. The overall performance of APL is however comparable to the CUDA C with the exception of the Matrix Multiplication benchmark which is up to 64x faster with CUDA C with regards to the kernel runtime.

We implemented five optimizations to help fulfill this requirement, but these did not provide the same level of kernel performance, as is the case with the CUDA C implementation.

But since we achieved better performance on at least two of the benchmarks in steady state in Section 5.3, we rate Efficiency as partially fulfilled, but not fulfilled since APL does not produce as efficient a PTX kernel as the handwritten CUDA C implementation.

Scalability - Important - Partially fulfilled *How well the implementation scales with regard to the input*
For scalability we have three requirements:

1. APL implemented benchmarks must scale at least as well as the sequential for-loop, TPL and CUDA C implementation.
   In Section 5.4 we see that APL and CUDA C implementations scale equally well, except for Matrix Multiplication for large inputs. APL scales at least as well as the sequential for-loop and TPL implementations.

2. APL must support all input sizes below the maximum number of threads runnable per device. APL has been hard-coded with a Cooperative Thread Arrays (CTA) size of 512, where compute capability 2.0 devices support CTA sizes of up to 1024. This limits us to half the number of thread possible on our device.

3. APL must handle as much data as there is memory available on the device. Our benchmarks are run with up to 629 MB of data, during which we have not experienced any limitations to the amount of memory available to APL.

Because APL does not scale as well as the CUDA C Matrix Multiplication implementation, we only support half the number of thread possible on compute capability 2.x, and we have not tested memory up the maximum available, we rate scalability partially fulfilled.

Correctness - Important - Fulfilled *Fulfilling the formalized requirements*
We have tested the correctness of APL by running the benchmark implementations and comparing the results. As stated in Chapter 5, all the APL benchmarks produce the same results as the other implementations. Thus we rate this requirement fulfilled.

Comprehensibility - Important - Fulfilled *The effort of ensuring a coherent understanding of the system*
To facilitate comprehensibility, we required that the implementation should be documented through Unified Modeling Language (UML) diagrams and the implementation should be described. This is done in Chapter 4. We thus rate this requirement fulfilled.

Testability - Important - Fulfilled *The expense of ensuring that the system fulfills the requirements*
We rate this requirement fulfilled since we have implemented a benchmark runner, as described in Appendix D, which implements the required features: Is able to run all benchmarks on all implementation, output timings and verify the correctness of the benchmarks calculations.

Interoperability - Less Important - Fulfilled *The expense of coupling the system with other systems*
The system should interoperate with the necessary parts of CLI to run the benchmarks implementations. We rate this requirement fulfilled, as we are able

to run the implemented benchmarks on APL and generate the correct result, thus APL is able to interoperate with the required parts of CLI.

Usability - Less Important - Fulfilled *Adaption to the technical-, work- and organizational environment*
APL has the same interface as TPL, which was the only requirement for fulfilling usability. We thus rate it fulfilled.

Reusability - Less Important - Fulfilled *The ability of using parts of the system in other systems*
We have divided the APL source code in components with different responsibility, as described in Section 4.2. This facilitates that parts of code can be exchanged, e.g. to handle other target languages. Thus we rate reusability fulfilled.

Maintainability - Less Important - Fulfilled *The expense of finding and correcting errors in the system*
To facilitate maintenance we split the code into components by responsibility. This increases cohesion and loose coupling between components, thus it is simpler to make changes to parts of the code, as changes does not propagate. We therefore rate maintainability as fulfilled.

Reliability - Less Important - Partially fulfilled *Fulfillment of the required level of precision*
The requirements for fulfilling reliability is that APL should be able to calculate the correct result for each of the implemented benchmarks. In addition, APL must support exception handling. The benchmarks does produce the correct result, but exception handling has however not been implemented, and we thus rate reliability as only partially fulfilled.

Functional Requirements

Our benchmark serves as an acceptance test for our functional requirements in Section 4.1, that is, if all benchmarks are able to run and produce the same result as the other benchmark implementations, the APL fulfills the acceptance test. By running our benchmark runner, described in Appendix D, we have tested all benchmarks and that they produce the same result. We thus conclude that the functional requirements have been fulfilled.

6.2 Discussion

The aim of this section is to discuss our choices and results, specifically, we will look at our choice of development method, whether we chose a good benchmark suite or not, and our choices of technologies. Finally, we will discuss the learnings goals that we presented in Section 2.3.1, discuss a problem we experienced with 64bit pointers on the device, and conclude the discussion with a discussion of the usefulness of APL.

6.2.1 Development Method

As described in Section 2.4, we have used an incremental development method which has been a great help for us, since it allowed us to freeze development of the code when we got close to the deadline of the project, despite not having implemented all the features and optimizations that we would have liked to. These features and optimizations are described in Section 6.3.

The automated tests and benchmarks was a great help doing the development, especially when implementing optimizations since we could ensure that we did not break previously working functionality, and easily see the effect the optimizations had on performance.

Benchmark Suite

During this project, we have implemented a benchmark suite which contains four benchmarks implemented in C# using sequential for-loops, C# using parallel loops from TPL, C# using parallel loops for APL and C++ using the CUDA Runtime API with CUDA C. This benchmark suite is however rather limited and we would have liked to have included more benchmarks if we had more time, since it do not represent all types of workloads.

It would have been interesting to have a benchmark which made use of more registers, since none of the benchmarks in our current benchmark suite exceed the 20 registers per thread, which according to the CUDA occupancy calculator a compute capability 2.0 GPUs can run without loosing any of the latency hiding ability. Therefore, it would be interesting to run a larger benchmark like the Ray Tracer, implemented on the 9th semester, which made use of 39 registers [16, sec. 4.2.3].

All of our current benchmarks make uses of sequential memory accesses which perform better on the GPU than non-sequential accesses. Therefore it would be interesting to have a benchmark which relied more on unsequential memory accesses, this could for example be the Pruned Neighbor Search we implemented on the 9th semester as part of our Boids model [16, sec. 3.1.2.2].

Few of the iterations in any of the benchmarks we currently have, take different branches. It would be interesting to have a benchmark which did more branching since this can have a large impact on the performance of a GPU application, as each warp can only follow a single thread at a time. A benchmark which could be used for this is the odd-even transposition sort we implemented on the 9th semester [16, 3.1.2.1].

As a whole our benchmark suite is not sufficiently varied to eliminate bias from the results. However, the differences we see in the performance of APL and TPL are in most cases so large that it is unlikely that they can be caused by bias. We do see small differences between some of the optimizations in which bias might have some effect, and cause us to draw incorrect conclusions.

6.2.2 Choice of Technologies

Based on the analysis in Chapter 3 we chose to use CIL as the source language of APL and JIT compile it to PTX using SSA. SSA was chosen to help us translate from the stack based representation of CIL to the register based representation of PTX. In the following, we will reflect on these choices.

We chose to use CIL as the source language of APL since this allows APL to be used in multiple high level languages and allows libraries, where the high-level source code is not available, to be used along with APL. We still find CIL to be a good choice, since CIL is the most flexible solution and since CIL is a relatively simple language. This simplicity means that we were not required to implement a parser for all the features of the high-level language, but could just create a relatively simple parser for the CIL using SSA. The `Reflection` namespace is used as it enabled us to access CIL at runtime.

We decided to use JIT compilation to translate from CIL to PTX, which we still find to be a good choice since it allows APL to be used without making changes to existing compiler tool chains. Furthermore, the benchmarks shows the time spend on JIT compiling is very low, as seen in Section 5.2.3, at least for the benchmarks we use.

Based on the decision to use CIL as the source language and perform JIT compilation, we chose to use PTX as the target language. We still find this to be a good choice since we avoid having to decompile the low-level CIL code to a high-level language such as CUDA C or OpenCL C. We also avoid having to deal with redistributing of *nvcc*, which is required to JIT compile CUDA C, and we do not have to worry about not being able to take advantages of all the new features introduced in Compute Capability 2.x, which are not available in OpenCL C.

If we had chosen to use a high-level source language, it would have been better to target CUDA C or OpenCL C, as this would have allowed us to take advantage of the optimizations already implemented in *nvcc* and Nvidia's OpenCL C compiler implementation, and thereby possibly achieving our efficiency goal. For example, in the case of the Black Scholes benchmark we expect that the performance difference we see is caused by *nvcc* performing more optimizations than we do.

We use reflection to access the instructions of an `Action`. This allows us to overcome the limitation of `Expression` only being able to generate expression trees for Expression Lambdas. While we could have implemented a parser which generated an expression tree, we expect this approach would have been more time consuming than our approach of using SSA and reflection to generate our own IR.

Non-functional Requirements Table 6.1 shows the result of our conclusion upon the non-functional requirements in Section 6.1.2. We have managed to fulfill three important and four less important non-functional requirements. One important requirement, one very important and a less important requirement is rated partially fulfilled. The very important requirement, Efficiency, required that APL produced kernels which had the same running time as the CUDA C implementation, this however proved to be difficult, and none of our results showed equivalent or better performance than CUDA C kernels. We did however beat the overall runtime of CUDA C on two benchmarks in steady state.

One reason for this is that CUDA C allows for more complex optimizations than APL, e.g. Matrix Multiplication splits the data into several small blocks, which is put in shared memory before they are multiplied which gives a 64x speedup with regards to the kernel runtime compared with APL. By allowing the user to nest several APL loops and implement an optimizations which uses shared memory where possible, it may be possible to use the same algorithm as the CUDA C implementation does and thus achieve the same kernel time.

Furthermore the efficiency requirement state that APL should achieve a total runtime comparable to the CUDA C implementation. Some of the benchmarks, e.g. Matrix Multiplication, can be said to not achieve this, as the CUDA C implementation is much faster than the APL implementation.

Scalability required that we: 1. scale as well as the other implementations, 2. allow for all input sizes up to the maximum number of threads runnable on the device and 3. that APL must handle data up to size of memory available on the device. We did however only support half of the threads possible, because we initially targeted 1.x, which only supports 512 threads in a thread block. This is easily fixable, as it will not affect other parts of the code. Furthermore we only tested APL with benchmarks of up to 629 MB of data. We did however not see any problems so far, and there is nothing in the APL implementation to stop us from using more data on the device. With regards to reliability, exception handling has to be added to APL to fulfill this requirement, this is covered in more detail in Section 6.3.2. Based on these reflections, we argue that despite three requirements not being fully fulfilled, the solution is still usable, and will in many cases give the user a speedup over sequential and parallel CPU implementations of code.

	Not fulfilled	Partially fulfilled	Fulfilled
Efficiency (V)		✓	
Scalability (I)		✓	
Correctness (I)			✓
Comprehensibility (I)			✓
Testability (I)			✓
Interoperability (L)			✓
Usability (L)			✓
Reusability (L)			✓
Maintainability (L)			✓
Reliability (L)		✓	

Table 6.1: Result from conclusion on non-functional analysis. V means the requirement is rated very important, I means rated important and L means less important.

6.2.3 Learning Goals

In Section 2.3, we formulated a list of four learning goals. Below we will discuss these and their relevance.

Gain knowledge of the .NET framework and its architecture In order to implement APL we needed to gain insight to the .NET framework and its architecture. We have documented this insight through the analysis, i.e. Section 3.2 talks about CLI, Section 3.6 talks about CIL and Section 3.8 talks about the possible ways of translating CIL to PTX. This knowledge proved useful during development, e.g. we knew how to extract CIL code from an `Action` delegate method.

Gain knowledge of the Nvidia CUDA compiler and toolchain In Section 3.4, we found that Nvidia's official CUDA C compiler, `nvcc`, allows the programmer to compile both host and kernel code. In addition, we found that *nvcc* produces PTX code which is embedded into the application. We also found that *nvcc* supports AOT compilation of PTX to binary device code, which is embedded along with the PTX code. We also learned that the CUDA Driver API can load PTX code at runtime, which is then JIT compiled to the specific GPU and invoked. Looking at OpenCL C, we see that the OpenCL API supports loading of OpenCL C kernels and JIT compiling these to the device. This is not supported by the CUDA Driver API, thus CUDA C cannot be loaded at runtime and JIT compiled.

Gain knowledge of the new features of Compute Capability 2.0 Appendix E contains an analysis of the differences between compute capability 1.0 and 2.0. We did however not get far enough with the implementation of APL to utilize many of the new features in compute capability 2.0, such as recursive functions or classes. One thing we did use from compute capability 2.0 is the L1 cache, as described in Section 5.2, which lead to speedup in some cases.

Gain experience with code translation Given that we have implemented a compiler which translates CIL to PTX, we argue that we have gained experience with the topic of code translation.

6.2.4 64bit On Device

Recall from Section 4.5.2 that we produced PTX code which utilized 64bit pointers at the start of the project, due to our device having 64bit addressing support, but that we later changed the `Emitter` such that 32bit pointers were used instead. The reason was that the CUDA Driver API threw "unknown" CUDA errors, i.e. errors which CUDA could not classify, and the results produced by the kernel was at times corrupted.

This behavior was first observed when we used "u64" types as pointer into global memory, and issued a `st.global` instruction which writes a value to the address pointed to by the 64bit pointer, using a GeForce GTX 470 GPU. We also tried a Quadro 880M with compute capability 1.2. In this case, the use of 64bit pointers was not a problem, even though the 880M does not support 64bit addressing, whereas the GTX 470 does.

Looking at the PTX produced by *nvcc*, we see that 64bit pointers are also used. Thus, we think that the problem is not with the PTX kernel code, but might be with the code which invokes the kernel.

This behavior might be caused by the different memory alignment requirements, i.e. using 32bit requires that memory is 32bit aligned while 64bit requires that memory is 64bit aligned. This would also explain why the kernel works correctly on the 880M device, since the 880M has no 64bit support and thus simply uses the 64bit registers as a 32bit pointer where the last 4 bytes are not used.

On the GTX 470, the `st.global` will initiate a 64bit transfer using a non-naturally aligned address that might fail. The PTX documentation states that the "address must be naturally aligned to a multiple of the access size" [45, Table 86], and if the address is not properly aligned, the resulting behavior is undefined, which is the behavior we see.

Thus, to fix this problem in future iterations of APL, we must look at the memory alignment requirements and implement proper alignment.

6.2.5 Potential Problems With Copy Omit

As mentioned in Section 4.4.2 the Copy Omit optimization can potentially cause APL to produce incorrect results, this can happen in the case where the host creates an array and writes some data to it before the array is then used as write only by APL, since the data written to the array by the host is not preserved after the call to APL. In our benchmark suite, this was not a problem since all the benchmarks which uses write only arrays writes to all positions in these. In future iterations of APL, this should however be taken into account, be it by detecting if the host writes data to the array prior to a call to APL, or by creating a mask on the device and only overwriting the positions in the host array that has been changed on the device, as described in Section 4.4.2.

6.2.6 Is APL Useful?

Looking at the abstraction provided by APL, and comparing this abstraction to CUDA C and OpenCL C, we argue that APL provides a higher abstraction level due to APL abstracts away from the concepts of data marshalling and launch configuration, and is thus easier for programmers to use. In addition, programmers, which are familiar with TPL, can move their computations to the GPU by changing only a few lines of code. Doing the same, using CUDA C and the CUDA Runtime API, requires several changes to the existing application.

Currently, it might however be necessary to make some changes to existing TPL programs to achieve higher performance, due to the APL compiler not optimizing the kernel as much as possible. This can be seen in the Matrix Multiplication benchmark in Section C.3, which as shown have modified from the TPL version to run more thread and use variables as an accumulator to avoid accesses to global memory. These two changes could however have been avoided by implementing the optimizations described in Section 6.3.1, and Section 6.3.1 respectively in the APL compiler.

The abstraction provided by APL is that of parallel loops, which we have seen in Section 2.2 is also provided by hiCUDA. hiCUDA is however relatively low level, since the programmer must explicitly use directives to specify which data is marshalled to the device and how the parallel-loop is broken up into thread blocks. These low-level directives does however provide the programmer with more control, which might improve kernel performance in some situations.

Comparing APL to GPU.NET, we see that GPU.NET only provides a thin abstraction on-top of CUDA C. Kernels are written as a .NET method and tagged with an attribute. The methods are later compiled to device code and executed on the device. Even though GPU.NET removes the aspect of manually marshalling data when writing CUDA C applications, it is still required that the programmer specify the thread and thread-block sizes. This is not required in APL, thus APL provides a higher level of abstraction.

Comparing APL with Accelerator, we see that APL is more general purpose while Accelerator is more domain specific for parallel array computations. Even though the for-loop abstraction provided by APL works well using arrays, we are allowed much greater freedom with access patterns such as random reads and writes. In addition, Accelerator requires that the programmer must make use of the Accelerator provided types, where operations on these are carried out on the device. APL does not impose such requirements, the only requirement of APL is that the programmer must parallelize a for loop using the `Parallel.For` method.

Looking beyond prior work from the start of the this project, we see that several new publications have been made by researchers on the topic of GPGPU.

A publication from April 2011 [49] argues that interpreted languages have failed to utilize modern processors, specifically the Single Instruction, Multiple Data (SIMD) aspect of modern processors. Looking at our project, we see that we are essentially trying to solve the same problem, which is to utilize modern processing power in an interpreted/JIT compiled environment. [49] argues that improving SIMD utilizations is important since GPU components, such as wide SIMD arrays, are finding its way to the CPU. This is already the case with the AMD fusion processors.

A publication from March 2011 [15] deals with optimizations techniques, specifically optimizations to reduce branch divergence in GPU programs, which can improve performance. APL does not perform any optimizations on branches. Their performance improvements range from 12%-80%, and may therefore be considered for APL.

A publication from 2011 [5] works with an existing parallel extension to the C language, which is called the Unified Parallel C (UPC) language, that is aimed at programming large-scale parallel machines. [5] extends UPC with a new execution which is closer to the execution model of CUDA, thereby allowing UPC applications to run on the GPU. Our solution is very similar to their solution, except that our solution is meant for TPL applications were their solution is meant for UPC applications.

6.3 Future Development

The aim of this section is to cover some of the ideas that have come to mind through out the development of APL. We will first cover some of the optimizations that we thought about implementing for our current set of features, but did not find the time to do. Afterwards, we will cover a broader set of features that will make APL more useful.

6.3.1 Optimizations

Recall from Section 4.7 and Section 5.2 that we implemented five optimizations that provided some speedup. The aim of this section is to cover other optimizations, which we though about during development of APL, but we did not have time to implement, and which might provide even more speedup.

```
Parallel.For(0, size, delegate(int i)
{
  v2[i] += v1[i];
});
Parallel.For(0, size-1, delegate(int i)
{
  result[i] = v2[i] + v2[i+1];
});
float foo = result[2];
```

Figure 6.1: Illustration of how data transfer between host and device is not always needed

CUDA Occupancy Calculator

The launch configuration of a kernel has an impact on how many threads can run concurrently on each Streaming Multiprocessor (SM) and thereby the performance of the kernel. Nvidia provides an Excel document called the CUDA Occupancy Calculator, which makes it easy to calculate how many threads can be executed concurrently on each SM on the GPU. The number of threads which can run concurrently depends on the compute capability of the device, how many registers the kernel uses, how much shared memory the kernels uses and the size of the CTAs, that is the number of threads in a thread block.

We can implement the CUDA Occupancy Calculator into APL and use this to determine the optimal CTA size, and thus maximize the occupancy rate of a kernel. This means that when the APL compiler has produced the PTX kernel, the integrated occupancy calculator can be run and the CTA size can be determined and the launch can be configured with this CTA size in mind.

Copy back to host on access

There is no need to copy data back to the host from the device until the host accesses the data. If the copying of data is delayed until the data was actually needed, data transfers could be avoided in cases where the data is never used on the host. Another case is illustrated in Figure 6.1, where the data is not used by the host between two calls of APL, and thus the data can be kept on the device between the two calls, thereby removing the data transfer overhead.

Nested parallel loops

GPUs require a lot of threads to achieve high performance, i.e. the GeForce GTX 470 available to us must have at least 21504 threads running concurrently to achieve the best possible performance, since this device has 14 SMs and each SM can have up to 1536 resident threads [47, p. 111][47, p. 154]. Therefore, it might be advantageous to use nested parallel loops to create additional threads as shown in Figure 6.2, where the APL kernel is equivalent to the CUDA C program shown on Figure 6.3. The CUDA C code runs $size^2$ threads rather than just *size*. This approach to implement nested loops is however somewhat limited since there is no way of synchronizing threads across multiple CTAs, without running multiple kernels, and it is therefore not possible to guarantee

```
Parallel.For(0, size, delegate(int y)
{
  Parallel.For(0, size, delegate(int x)
  {
    result[x + y * size] = x + y;
  });
});
```

Figure 6.2: Example of nested parallel loops in APL

```
__global__ void example (float* result, int size)
{
  int x = blockDim.x * blockIdx.x + threadIdx.x;
  int y = blockDim.y * blockIdx.y + threadIdx.y;
  if( x < size && y < size)
  {
    result[x + y * size] = x + y;
  }
}

public void hostExample()
{
...
...
dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
dim3 grid(size / BLOCK_SIZE, size / BLOCK_SIZE);
example<<< grid, threads >>>(d_result, size);
...
...
}
```

Figure 6.3: Example of how a nested parallel loops in APL might look in CUDA C

that operations in the outer loop is performed in the correct order. In Section 6.3.1, another approach to handling nested loops is shown, which overcomes this limitation but instead imposes limits on the size of the nested loop.

Local objects

If data is allocated inside a parallel loop and operated on by nested parallel loops, as is done in Figure 6.4, it could increase performance if the data were allocated in shared memory. Shared memory could be used by mapping each iteration in the outer loop to a CTA and each iteration of a nested loop to a thread in the CTA as shown in Figure 6.5. To be able to do this, the data used in the outer loop must fit in the shared memory, which for compute capability 2.0 devices is 48KB, and as the iterations of the nested loops are mapped to threads in a CTA the maximum number of iterations which can be performed by each nested loop is limited to 1024, since 1024 is the maximum number of threads in a CTA on a compute capability 2.0 device. Because it is possible to synchronize threads using PTX synchronization instructions such as `bar.sync`, it possible to do multiple operations in the outer loop and still guarantee the correct order of execution.

```
int bar[] = new int[sizeX * sizeY];
Parallel.For(0, sizeX, delegate(int x)
{
  int[] foo = new int[sizeY]; //allocated in shared memory
  Parallel.For(0, sizeY, delegate(int y)
  {
    foo[y] = x;
  });
  Parallel.For(0, sizeY, delegate(int y)
  {
    if(y-1 >= 0)
    {
      bar[x + y * sizeX] = foo[y-1];
    }
  });
});
```

Figure 6.4: Example of how a nested parallel loops can operate on a object allocated in the outer loop

```
__global__ void example (int* bar, int sizeX ,int sizeY)
{
  int x = blockIdx.x;
  int y = threadIdx.x;
  __shared__ int* foo;

  if (threadIdx.x == 0)
  {
    foo = (int*)malloc(sizeY * sizeof(int));
  }
  __syncthreads();
  foo[y] = x;
  __syncthreads();
  if(y-1 > 0)
  {
    bar[x + y * sizeX] = foo[y-1];
  }
}
public void hostExample()
{
  ...
  ...
  example<<< sizeX, sizeY >>>(d_bar, int sizeX, int sizeY);
  ...
  ...
}
```

Figure 6.5: Example of how local objects can be allocated in shared memory in CUDA C

```
int size = 1000;
Parallel.For(0, size, delegate(int i)
{
  result[i] = 0;
  for (int j = 0; j < size; j++)
  {
    result[i] += m1[i,j];
  }
});
```

Figure 6.6: Example of multiple accesses to the same array element

Asynchronous GPU execution

There is no need for the device and the host to run synchronously at all times. The host could continue executing after a call to APL, and only synchronize with the device when accessing a variable which is written to by the device. As with the optimization described in Section 6.3.1, this could lead to a increase in performance as the device and the host would be capable of working on different tasks in parallel.

Constant and Texture memory

In our 9th semester project, we found that the use of other on chip memory, other than shared memory, can lead to a substantial increase in performance [16, sec. 4.2.2]. There are some possibilities for APL to take advantage of the different types of on chip memory, such as constant memory and texture memory, there can lead to an increase in performance.

If data is only read from, it might give better performance if it is allocated in one of the read only memory spaces. The constant memory space can provide access to an on chip cache. In addition to being able to broadcast a read from a single memory address to multiple threads, the texture memory space also gives access to an on chip cache which is optimized for spatial locality. The texture memory cache is however higher latency than the L1 cache and might hurt performance in some situations where the memory accesses can be better cached by the L1 cache.

Cache Memory Access in Registers

In some case the same element in a array is accessed multiple times without any explicit synchronization between them as e.g. seen in Figure 6.6 where `results[i]` is accessed 2001 times for each iteration of the parallel loop. Since each access to a array element corresponds to a global memory access it might increases performance if we instead cached the array element in a register. Since TPL and therefore also APL does not provide any guarantees about the order of execution of the loop iterations, or how many iterations run concurrency, such caching will not effect the correctness of the program since any problems with it would be as a result of a race condition which might give problem even with no caching support.

6.3.2 Features

In this section, we cover some of the features which came to mind while implementing APL, but we did not have time to do, or was beyond the scope of this project. We will consider two sets of features, the need to have features and the nice to have features.

Need to Have

In this section, we will cover all features which we argue needs to be implemented prior to an official release of APL.

All OpCodes The `Parser` of APL currently supports 46 opcodes, which is roughly 1/4th of the opcodes found in CIL. This means that many of the language construct in CIL cannot be translated to PTX and executed on the device, due to lacking opcodes.

Implementing support for all opcodes would remove this problem, since APL would then able to express all CIL instructions in PTX. This would mean that higher level abstractions can be supported in languages such as C#, since C# compiles to CIL.

Many of the lacking opcodes are easy to implement, since many of them are alternatives to opcodes which APL already supports, e.g. several of the opcodes deal with branching which are very similar to implement. Some opcodes are however harder to implement, e.g. the `newarr` opcode, which allocates an array. Currently we have no support for memory allocation, this must be implemented prior to support for this particular opcode.

Structures We have support for arrays of type `System.Single` and `System.Int32`, and their element types. Support for other primitive data types is easy to implement, e.g. a `System.Int16` can be mapped to the PTX type `.s16`.

Support for complex types such as user defined structures was beyond the scope of this project. Support for these are however a necessity in future iterations of APL, since structures allow the programmer to group primitive types into records which can be passed around, thus increasing the abstraction level. Implementing support for structures is not a big challenge, since structure support is also available in CUDA C, even for 1.0 compute capable devices.

Classes Class support is a feature which we would like to see added to APL, because most languages of CLI support the concept of classes. Class support is however not a trivial matter to implement, since we must support features such as inheritance, interfaces, and virtual methods.

CUDA C does support C++ classes to some extent, and with the release of CUDA Software Development Kit (SDK) 4.0, CUDA C's class support has been extended with virtual methods on compute capability 2.x devices. Thus, we are certain that PTX has support for the necessary features required to implement class support in APL.

Security Security aspects, such as preventing APL in taking over the device by a malicious application, or such as preventing crashes if array boundaries have been breached by the kernel, should have priority in future iterations of APL.

In the current state, a malicious program can overload the device using the `Parallel.For` method, which will make the system unresponsive or result in a blue screen due to the device being shutdown after a certain amount of time. Even in situations where the program is not malicious, the kernel might require more resources by the device than available, thus resulting in an unresponsive system.

Array boundary exceptions, or other types of exceptions, are frequently encountered in .NET applications. APL currently has no support for exceptions of any kind, not even array boundary checks. This means that the kernel might corrupt the memory on the device, which can result in artifacts in other programs running on the system. In real life applications, this exception handling is a necessity.

Nice to Have

In this section, we will cover the features which we would like to see added to APL, but is not required in the first official release of APL.

Dynamic Memory Currently, all data required by our produced kernels is preallocated prior to invocation. In some situations, it is desirable to allow dynamic memory allocation, e.g. if the output size is not known at compile time.

Dynamic memory allocation has been introduced using the `alloca` built-in function which is available in PTX 2.x. This function allocates local memory for the thread to use. It appears that there are no support for dynamic memory allocation of global memory, thus global memory must be allocated prior to kernel invocation.

CUDA C has a form of dynamic memory support, but this is done by allocating a big chunk of global memory, which is afterwards used by the other threads as a heap [47, sec. B.15]. We can adopt a similar technique, and allocate a big chunk of global memory prior to kernel invocation for the heap.

Debugging Support Currently, we have no debugging support in APL meaning that it is not possible to reach breakpoints which for example have been set through the Visual Studio IDE. To implement breakpoint support, the `break` opcode must be implemented, which signals the Virtual Execution System (VES) that a breakpoint has been reached.

The `break` opcode is implementation specific [17, 3.16], meaning that we are free to use this opcode as we see fit. With regards to PTX, the documentation does not mention debugging support other than the `brkpt`, `pmevent` and `trap` instructions, which are categorized as miscellaneous instructions. The `brkpt` instruction is used to indicate breakpoints and suspends the execution of the kernel, the `pmevent` is used to perform monitor events such as incrementing one of the four performance counters, and the `trap` instruction aborts the execution of the kernel. These instructions are vaguely documented and their usage is not clear.

However, we know that it is possible to debug CUDA C kernels when executing these on a device, if an independent device is used to render the Graphical User Interface (GUI) of the machine in which debugging is carried out[44]. And since CUDA C is translated to PTX, it should be possible for us to include the same debugging support as is supported in CUDA C.

Multiple Devices Currently, APL can offload computations to one device, but does not support offloading of computations to multiple devices present in the system. Introducing support for multiple devices allows APL to use the extra computational power present in the system. Utilizing multiple devices requires that we change much of APL, specifically, we must include support for multiple devices by changing the code that deal with the CUDA Driver API. In addition, each device has its own global memory, thus some form of synchronization between the devices must be implemented prior to being able to support this feature.

IDE Support for Overhead With the introduction of Visual Studio 2010, Microsoft has included support for a graphical tool, called the Parallel Performance Tool, used to indicate how well a TPL powered application performs on the CPU. This tool allows the developer to analyze how well his parallel program performs, i.e. how much each core is utilized and the amount of time each thread is blocked. [50]

Introducing support for APL in the Parallel Performance Tool allows the developer to determine how well his parallel program runs on the device, i.e. by showing the percentage of active threads and the number of uncoalesced memory accesses. Another possibility is to introduce this support into the Nvidia Parallel Nsight extension to Visual Studio, which allows debugging and performance profiling of CUDA kernels.

Bibliography

[1] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 1–11, New York, NY, USA, 1988. ACM.

[2] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. *SIGPLAN Not.*, 41:169–190, October 2006.

[3] Stephen M. Blackburn, Kathryn S. McKinley, Robin Garner, Chris Hoffmann, Asjad M. Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovik, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. Wake up and smell the coffee: evaluation methodology for the 21st century. *Commun. ACM*, 51:83–89, August 2008.

[4] Simone Campanoni, Giovanni Agosta, and Stefano Crespi Reghizzi. A parallel dynamic compiler for cil bytecode. *SIGPLAN Not.*, 43:11–20, April 2008.

[5] Li Chen, Lei Liu, Shenglin Tang, Lei Huang, Zheng Jing, Shixiong Xu, Dingfei Zhang, and Baojiang Shou. Unified parallel c for gpu clusters: Language extensions and compiler implementation. In Keith Cooper, John Mellor-Crummey, and Vivek Sarkar, editors, *Languages and Compilers for Parallel Computing*, volume 6548 of *Lecture Notes in Computer Science*, pages 151–165. Springer Berlin / Heidelberg, 2011.

[6] Morten Christiansen and Christian E. Hansen. CUDA DBMS. https://services.cs.aau.dk/public/tools/library/files/rapbibfiles1/1244627230.pdf. Last seen: 14th of March.

[7] Cristina Cifuentes, Doug Simon, and Antoine Fraboulet. Assembly to high-level language translation. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 228–, Washington, DC, USA, 1998. IEEE Computer Society.

[8] Hoopoe Cloud. Cuda.net. http://www.hoopoe-cloud.com/Solutions/CUDA.NET/Default.aspx. Last seen: 9th of April 2011.

[9] DotGNU. DotGNU Portable.NET. http://www.gnu.org/software/dotgnu/pnet.html. Last seen: 21th of February.

[10] ECMA. Standard ecma-335 - common language infrastructure (cli). http://www.ecma-international.org/publications/standards/Ecma-335.htm. Last seen: 21th of February.

[11] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. *SIGPLAN Not.*, 42:57–76, October 2007.

[12] GPGPU.org. The official gpgpu faq. http://www.gpgpu.org/wiki/FAQ. Last seen: 23th of May.

[13] Marisabel Guevara, Chris Gregg, Kim Hazelwood, and Kevin Skadron. Enabling task parallelism in the cuda scheduler. *Work*, page 69076, 2009.

[14] Tianyi David Han and Tarek S. Abdelrahman. hicuda: High-level gpgpu programming. *IEEE Trans. Parallel Distrib. Syst.*, 22:78–90, January 2011.

[15] Tianyi David Han and Tarek S. Abdelrahman. Reducing branch divergence in gpu programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 3:1–3:8, New York, NY, USA, 2011. ACM.

[16] Søren Alsbjerg Hørup, Søren Andreas Juul, and Henrik Holtegaard Larsen. *The Art of GPGPU.* Aalborg University, 2011.

[17] ECMA International. *Standard ECMA-335 - Common Language Infrastructure (CLI).* ECMA International, 5 edition, June 2010.

[18] David M. Lane. Online statistics education: An interactive multimedia course of study. http://onlinestatbook.com. Last seen: 14rd of Marts 2011.

[19] Seyong Lee and Rudolf Eigenmann. Openmpc: Extended openmp programming and tuning for gpus. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[20] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. Openmp to gpgpu: a compiler framework for automatic translation and optimization. *SIGPLAN Not.*, 44:101–110, February 2009.

[21] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. *SIGARCH Comput. Archit. News*, 38:451–460, June 2010.

[22] David J. Lilja. *Measuring computer performance: a practitioner's guide*. Cambridge University Press, New York, NY, USA, 2000.

[23] Lars Mathiassen, Andreas Munk-Madsen, Peter Axel Nielsen, and Jan Stage. *Objektorienteret analyse & design*. Marko, Aalborg, 2001.

[24] Microsoft. The .net framework. `http://www.microsoft.com/net/`. Last seen: 21th of February.

[25] MSDN. Common Language Runtime (CLR). `http://msdn.microsoft.com/en-us/library/8bs2ecf4.aspx`. Last seen: 21th of February.

[26] MSDN. Common type system. `http://msdn.microsoft.com/en-us/library/zcx1eb1e.aspx`. Last seen: 21th of February.

[27] MSDN. Expression Trees (C# and Visual Basic). `http://msdn.microsoft.com/en-us/library/bb397951.aspx`. Last seen: 21th of April 2011.

[28] MSDN. How to: Write a parallel.for loop that has thread-local variables. `http://msdn.microsoft.com/en-us/library/dd460703.aspx`. Last seen: 3rd of Marts 2011.

[29] MSDN. Lambda expression. `http://msdn.microsoft.com/en-us/library/bb397687.aspx`. Last seen: 17th of May 2011.

[30] MSDN. Module class. `http://msdn.microsoft.com/en-us/library/system.reflection.module.aspx`. Last seen: 28th of February.

[31] MSDN. Native image generator (ngen.exe). `http://msdn.microsoft.com/en-us/library/6t9t5wcf(v=vs.80).aspx`. Last seen May 30rd, 2011.

[32] MSDN. .NET Framework Conceptual Overview. `http://msdn.microsoft.com/en-us/library/zw4w595w.aspx`. Last seen: 21th of February.

[33] MSDN. Parallel class. `http://msdn.microsoft.com/en-us/library/system.threading.tasks.parallel.aspx`. Last seen: 3rd of Marts 2011.

[34] MSDN. Parallel.for<tlocal> method (int32, int32, func<tlocal>, func<int32, parallelloopstate, tlocal, tlocal>, action<tlocal>). `http://msdn.microsoft.com/en-us/library/dd783299.aspx`. Last seen: 3rd of Marts 2011.

[35] MSDN. Reflection. `http://msdn.microsoft.com/en-us/library/f7ykdhsy.aspx`. Last seen: 17th of May 2011.

[36] MSDN. What is the common language specification? `http://msdn.microsoft.com/en-us/library/12a7a7h3(VS.71).aspx`. Last seen: 22th of February.

[37] Mike Murphy. Tutorial on nvidia's open64 sources. http://wiki.open64.net/images/1/10/Nvopencc-tutorial.pdf.

[38] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! *SIGPLAN Not.*, 44:265–276, March 2009.

[39] Trey Nash. C# and the clr. In *Accelerated C# 2008*, pages 9–15. Apress, 2007. 10.1007/978-1-4302-0338-4_2.

[40] Novell. The Mono runtime. http://mono-project.com/Mono:Runtime. Last seen: 21th of February.

[41] Nvidia. Nvidia cuda - fermi compatibility guide for cuda applications. http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/Fermi_Compatibility_Guide.pdf. Last seen: 31th of Marts.

[42] Nvidia. Nvidia CUDA Library: cuFuncGetAttribute. http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/online/group__CUDA__EXEC_g5e92a1b0d8d1b82cb00dcfb2de15961b.html#g5e92a1b0d8d1b82cb00dcfb2de15961b. Last seen: 19th of April 2011.

[43] Nvidia. Nvidia CUDA Library: cuModuleLoadDataEx. http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/online/group__CUDA__MODULE_g9e8047e9dbf725f0cd7cafd18bfd4d12.html#g9e8047e9dbf725f0cd7cafd18bfd4d12. Last seen: 19th of April 2011.

[44] Nvidia. Parallel Nsight Requirements. http://www.nvidia.com/object/parallel-nsight-requirements.html. Last seen: 23th of May.

[45] Nvidia. PTX: Parallel Thread Execution - ISA Version 2.1. http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/ptx_isa_2.1.pdf. Last seen: 21th of March.

[46] Nvidia. The CUDA Compiler Driver NVCC. http://sbel.wisc.edu/Courses/ME964/2008/Documents/nvccCompilerInfo.pdf. Last seen: 31th of March.

[47] Nvidia. *NVIDIA CUDA C Programming Guide (Version 3.2)*. Nvidia, 2010.

[48] Nvidia. *Tuning CUDA Applications for Fermi (Version 1.3)*. Nvidia, 2010.

[49] Jonathan Parri, Daniel Shapiro, Miodrag Bolic, and Voicu Groza. Returning control to the programmer: Simd intrinsics for virtual machines. *Commun. ACM*, 54:38–43, April 2011.

[50] Hazim Shafi. Parallel Performance Tools . http://blogs.msdn.com/b/hshafi/archive/2009/05/18/visual-studio-2010-beta-1-parallel-performance-tools.aspx. Last seen: 18th of May.

[51] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program gpus for general-purpose uses. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 325–335, New York, NY, USA, 2006. ACM.

[52] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program gpus for general-purpose uses. *SIGOPS Oper. Syst. Rev.*, 40:325–335, October 2006.

[53] Techarp. New directx 11 features for the compute shader. `http://www.techarp.com/showarticle.aspx?artno=637&pgno=4`. Last seen: 23th of May.

[54] TidePowered. GPU.NET – develop GPU-accelerated applications with .NET. `http://www.tidepowerd.com/`. Last seen: 9th of March.

[55] TidePowered. GPU.NET - FAQ - Frequently Asked Questions. `http://help.tidepowerd.com/kb/general/faq-frequently-asked-questions`. Last seen: 9th of March.

[56] TidePowered. GPU.NET - urrently Supported - hardware, languages and their restrictions. `https://tidepowerd.tenderapp.com/kb/getting-started-with-gpunet/currently-supported-hardware-languages-and-their-restrictions`. Last seen: 9th of March.

[57] TidePowered. tidepowerd / GPU.NET-Example-Projects. `https://github.com/tidepowerd/GPU.NET-Example-Projects/blob/master/CSharp.BlackScholes/BlackScholes.cs`. Last seen: 9th of March.

[58] Wikipedia. Common language infrastructure — wikipedia, the free encyclopedia, 2011. [Online; accessed 4-June-2011].

A CUDA C and PTX test kernel

The code presented here is used in Section 3.4.1 and Section 3.8.3.

```
extern "C"
__global__ void mult(int *A, int *B, int *out, int max)
{
  int x = blockDim.x*blockIdx.x+threadIdx.x;
  if(x < max)
    out[x] = A[x] + B[x];
}
```

Figure A.1: Vector addition kernel in CUDA C

```
.version 1.4
.target sm_10, map_f64_to_f32
// compiled with /usr/local/cuda/open64/lib//be
// nvopencc 3.2 built on 2010-11-03

//-----------------------------------------------------------
// Compiling /tmp/tmpxft_00000e0b_00000000-9_t_k.cpp3.i (/tmp/
    ccBI#.YRfVEt)
//-----------------------------------------------------------

//-----------------------------------------------------------
// Options:
//-----------------------------------------------------------
//  Target:ptx, ISA:sm_10, Endian:little, Pointer Size:64
//  -O3 (Optimization level)
//  -g0 (Debug level)
//  -m2 (Report advisories)
//-----------------------------------------------------------

.file 1 "<command-line>"
.file 2 "/tmp/tmpxft_00000e0b_00000000-8_t_k.cudafe2.gpu"
.file 3 "/usr/lib/gcc/x86_64-linux-gnu/4.4.5/include/stddef.h"
.file 4 "/usr/local/cuda/bin/../include/crt/device_runtime.h"
.file 5 "/usr/local/cuda/bin/../include/host_defines.h"
.file 6 "/usr/local/cuda/bin/../include/builtin_types.h"
.file 7 "/usr/local/cuda/bin/../include/device_types.h"
.file 8 "/usr/local/cuda/bin/../include/driver_types.h"
.file 9 "/usr/local/cuda/bin/../include/surface_types.h"
.file 10 "/usr/local/cuda/bin/../include/texture_types.h"
.file 11 "/usr/local/cuda/bin/../include/vector_types.h"
.file 12 "/usr/local/cuda/bin/../include/
    device_launch_parameters.h"
.file 13 "/usr/local/cuda/bin/../include/crt/storage_class.h"
.file 14 "/usr/include/bits/types.h"
.file 15 "/usr/include/time.h"
.file 16 "/usr/local/cuda/bin/../include/texture_fetch_functions
    .h"
.file 17 "/usr/local/cuda/bin/../include/common_functions.h"
.file 18 "/usr/local/cuda/bin/../include/math_functions.h"
.file 19 "/usr/local/cuda/bin/../include/math_constants.h"
.file 20 "/usr/local/cuda/bin/../include/device_functions.h"
.file 21 "/usr/local/cuda/bin/../include/sm_11_atomic_functions.
    h"
.file 22 "/usr/local/cuda/bin/../include/sm_12_atomic_functions.
    h"
.file 23 "/usr/local/cuda/bin/../include/sm_13_double_functions.
    h"
.file 24 "/usr/local/cuda/bin/../include/sm_20_atomic_functions.
    h"
.file 25 "/usr/local/cuda/bin/../include/sm_20_intrinsics.h"
.file 26 "/usr/local/cuda/bin/../include/surface_functions.h"
.file 27 "/usr/local/cuda/bin/../include/math_functions_dbl_ptx1
    .h"
.file 28 "t_k.cu"
```

```
  .entry mult (
    .param .u64 __cudaparm_mult_A,
    .param .u64 __cudaparm_mult_B,
    .param .u64 __cudaparm_mult_out,
    .param .s32 __cudaparm_mult_max)
  {
  .reg .u16 %rh<4>;
  .reg .u32 %r<9>;
  .reg .u64 %rd<10>;
  .reg .pred %p<3>;
  .loc  28  2  0
$LDWbegin_mult:
  mov.u16    %rh1, %ctaid.x;
  mov.u16    %rh2, %ntid.x;
  mul.wide.u16  %r1, %rh1, %rh2;
  cvt.u32.u16   %r2, %tid.x;
  add.u32    %r3, %r2, %r1;
  ld.param.s32  %r4, [__cudaparm_mult_max];
  setp.le.s32   %p1, %r4, %r3;
  @%p1 bra  $Lt_0_1026;
  .loc  28  6  0
  cvt.s64.s32   %rd1, %r3;
  mul.wide.s32  %rd2, %r3, 4;
  ld.param.u64  %rd3, [__cudaparm_mult_A];
  add.u64    %rd4, %rd3, %rd2;
  ld.global.s32   %r5, [%rd4+0];
  ld.param.u64  %rd5, [__cudaparm_mult_B];
  add.u64    %rd6, %rd5, %rd2;
  ld.global.s32   %r6, [%rd6+0];
  add.s32    %r7, %r5, %r6;
  ld.param.u64  %rd7, [__cudaparm_mult_out];
  add.u64    %rd8, %rd7, %rd2;
  st.global.s32   [%rd8+0], %r7;
$Lt_0_1026:
  .loc  28  7  0
  exit;
$LDWend_mult:
  } // mult
```

Figure A.2: Result of compiling Figure A.1 to PTX with the `nvcc -ptx kernel.cu` command with CUDA compilation tools release 3.2

B Benchmarks

There are many different things to consider when measuring the performance of an application. In the following, we will look at some of these to form the basis for our approach to performing benchmarks as presented in Section 2.4.1. First, we will look at how bias might be introduced into performance measurements, and which techniques can be used to avoid this. Next, we will look at some of the things that should be taken into account when benchmarking applications implemented i managed languages such as Java or C#. Finally, we will show how to compute a confidence interval which can be used to indicate how confident we are in the correctness of our measurements.

B.1 Bias

As noted by [38], measurement bias is often a problem in publications dealing with performance measurements of computer systems. To avoid being bias towards one specific implementation, or optimization, and potentially draw wrong conclusions, [38] have investigated how such bias might be dealt with.

To illustrate that bias is both commonplace and significant, [38] conducted an experiment where they tried to find out whether compiling a C program with O2 or O3 optimization produces the fastest execution time. They did this by running a number of benchmarks from the SPEC CPU2006 benchmarks suite. To show how easy it is to introduce bias into benchmarks, they compile the benchmarks with different link orders and ran the benchmarks with different UNIX environment sizes. The link order can affect the performance of a program due to different code layout in memory, and the UNIX environment sizes, i.e. the number of bytes used to store environment variables, because it affects alignment of data allocated on the stack.

The results show that these small changes can have a significant impact on which conclusions are drawn based upon the performance changes. In one case, a benchmark compiled with O3 optimization gave a speed up of 1.1 compared to O2 optimization, while change in the link ordering gave a speedup of 0.92 with O3 over O2 optimization. Thus, one might incorrectly conclude that O2 is better than O3 or vise versa depending on the link ordering.

In [38], three approaches to handle the bias of a benchmark setup is presented. In the following sections, we will give an introduction to these.

B.1.1 Using a Diverse Benchmark Suite

By using a sufficiently diverse benchmark suite, any bias that might be present could be factored out, since the bias would effect the different workloads in the benchmarks differently. It is however hard to create a sufficiently diverse benchmark suite and benchmark suites might themselves be biased. It has e.g. been shown that the SPEC JVM98 benchmark suite is less memory intensive than real world Java applications [2], thus this benchmark suite might be biased against Virtual Machine (VM)s with sophisticated memory management. It has also been shown that the SPEC CPU2006 benchmark suite is not sufficiently diverse to factor out the bias introduced by different link orders and UNIX environment sizes [38, sec. 7.1.1].

B.1.2 Experimental Setup Randomization

By varying different parameters which are known to cause bias, and running the benchmarks multiple times, one can use statistical methods to factor out the effect of a bias in benchmark setups. To ensure that there is no bias present, it is important that each of the parameters is varied enough. It can however be hard to determine what is enough. It is required that a lot of benchmark runs are created, which might not be practical. In addition, it is requires that all parameters that affect the bias is know and it is often the case that we do not know all of these. [38, sec. 7.1.2]

B.1.3 Using Causal Analysis

Causal analysis can be used to ensure that we do not draw incorrect conclusions for our data, even if the data is affected by a bias in the measurements. The basic approach is that if we reach the conclusion that X caused Y, we can increase our confidence that Y is in fact caused by X, and not some bias, by first modifying X in some way, while having a minimal effect on all other parts of the system. Then use the modified version of X to measure the change to the system and finally see if Y changed in the way we expected, as a consequence of the changes to X. Thus, if a performance gain is seen when applying O3 optimization instead of O2 optimization, a change to the compiler, such that the O3 optimization is performed differently, could be made to increase the confidence that the increases in performance is caused by the O3 optimization, and not some lucky interaction between the optimization O3 and the benchmarks setup. [38, sec. 7.1.3]

B.2 Benchmarking Languages

As stated by [3], there are a number of things which should be taken into consideration when benchmarking Java applications, and managed languages in general. We will implement our benchmarks using C#, which is a managed language, we will therefore look at these considerations in the following. Note that the problems of choosing a Meaningful Baseline, Host Platform and Language Runtime is not exclusive to managed languages and should be taken into account when benchmarking unmanaged languages as well.

B.2.1 Meaningful Baseline

When performing benchmarks, it is important to find a meaningful baseline to compare ones implementation against, usually a meaningful baseline would be the current state of the art or the most widely used implementation. [3, sec. 3.3]

B.2.2 Host Platform

It is desirable to benchmark on different hardware platforms and different operating systems, since different implementations may exhibit different characteristic on different platforms. [3, sec. 3.3]

B.2.3 Language Runtime

The use of different compilers, VMs and libraries can affect the performance of an application, therefore, this should be considered when performing benchmarks. It should be ensured that all benchmarks in a benchmark run are using the same toolchain in order to make the results comparable. It is however desirable to have other benchmark runs which uses different toolchains, to investigate if that observed performance characteristics are present with other toolchains. [3, sec. 3.3]

B.2.4 Heap Size

Managed languages use garbage collectors to deallocate unused objects. When garbage collectors run, they free up memory but interfere with the normal execution of a program, thus causing the program to run slower, leading to a trade off between space and time usage. The heap size is a variable involved in controlling how often the garbage collector runs, i.e. the smaller the heap the more often the garbage collector will run. [3, sec. 3.2]

B.2.5 Warm-up

A single instance of a VM often executes the same part of an application multiple times, the first time the applications, runs a lot of warm up is involved due to JIT compilation of code, while later instances of the application involves less compilation and might even produce more optimized code. Given enough runs, the code will eventual reach a steady state where no further optimizations are applied. In real applications, steady state is the most common case, e.g. in applications servers the same code is run many times by the same VM instance. It might however sill be desired to measure the performance at startup by running each benchmark iteration in its own VM instance. [3, sec. 3.2]

B.2.6 Nondeterminism

Some high performance VMs try to measure how often methods are called and try to perform more optimization on the methods that are called often. Commonly, this is done using dynamic sampling, thus how much optimization i performed at a given point of execution can vary from run to run. To control this nondeterminism, three approaches are given: Replay Compilation, Multi-Iteration Determinism and Statistical Analysis. [3, sec 3.2]

Replay Compilation allows an application to run a number of training runs. The optimization plans created in the training runs are then recorded and replayed in subsequent timing runs, thus the same optimizations are performed on all timing runs. This however requires support by the VM. Most production VMs do not provide this support. [3, sec. 3.4] There is also a chance that replay compilation do not reflect the actual performance characteristic of the program running on a normal VM. [11, sec. 6.2.3]

Multi-Iteration Determinism does not require support from the VM, instead it runs iterations of a benchmark on a single instance of the VM until a steady state is reached and then measures the performance from that point on. When the steady state is reached can be determent by the coefficient of variance, defined as the standard deviation divided by the mean [22]. If this value falls below a threshold, such as 0.01 or 0.02 as suggested by [11], the iteration is considerate as being steady state. Thus measuring the performance of a benchmark over N iterations in steady state can be done by calculating the coefficient of variance over a window of N iterations and only stopping the benchmark run when the coefficient of variance falls below 0.02. As noted by [11] there is however still a chance that two different invocations of a VM may reach different steady states.

Statistical Analysis can be used to increase confidence in results when measuring in noisy environments by collecting data from multiple runs, but there is a limit to how many runs can practically be performed on a given benchmark, therefore, while it is possible to achieve a desired confidence in the results without using the two previous methods, it can be advantageous to use these to limit the noise and thereby require less data to achieve the same level of confidence. One way of measuring confidence in a result is by using a confidence interval.

B.3 Confidence Interval

The confidence interval for the mean is computed using a number of samples from a larger population. The confidence interval is an interval which is a certain probability, e.g. 90%, that the actual mean of the whole population lies within. If we want to increase the probability that the actual mean is in the confidence interval from e.g. 90% to 95% without changing the samples used to compute the interval we have to increase the size of the interval. [11, sec. 3.2.3]

Based on the confidence intervals of two alternatives we can compare these, if the two confidence intervals overlap we cannot say for certain that the differences seen in mean value is not due to noise in the measurements. If the two intervals do not overlap we can however conclude that there is no evidence to suggest that there is not a statistically significant difference between the two. That is, there is still some small chance that the difference between the two means might be caused by noise in the measurements. [11, sec. 3.3]

There exists a number of techniques which can be used to compare two or more alternatives as the ones presented in [11, sec. 3.4] and [11, sec. 3.5] we however find that these are beyond the scope of this project, so we will not present these. We will however show how to compute a confidence interval in Section B.3.1.

B.3.1 Calculating the Confidence Interval

The confidence interval around a mean can be computed based on a number of independent samples x_i, $1 \leq i \leq n$, taken from a population which actual mean is at μ. The goal when computing the confidence interval is to find a lower limit c_1 and a upper limit c_2 such that μ is in the range $[c_1, c_2]$ with a given probability. This probability is called the confidence level, and is chosen before computing the confidence interval. Thus we must estimate c_1 and c_2 such that $Pr\left[c_1 \leq \mu \leq c_2\right] = cl$ holds where Pr is an appropriate probability distribution, e.g. a normal distribution and cl is the confidence level. [11, sec. 3.2.1]

Computing the confidence interval around a mean can be done by using the general formula:

$$c_1 = M - (Z_{cl})(s_m)$$
$$c_2 = M + (Z_{cl})(s_m)$$

where M is the mean of the samples, s_m is the standard error for the mean given by the samples and Z_{cl} is the number of standard deviations from the mean required to cover cl percentage of the area of the probability distribution used. [18, sec. VIII.E.2]

The following describes how to calculate these in more detail: M is mean of the samples and can be calculated as:

$$M = \frac{\sum_{i=1}^{n} x_i}{n}$$

In order to calculate s_m the sample variance is first calculated [18, sec. VIII.B]

$$s^2 = \frac{\sum_{i=1}^{n}(x_i - M)^2}{n-1}$$

Then s_m is calculated as [18, sec. VIII.E.2]

$$s_m = \sqrt{\frac{s^2}{n}}$$

When calculation a confidence interval based on the sample variance, as is the case when calculating it based on performance measurements, the t distribution should be used rather than the normal distribution [18, sec. VIII.E.2]. This is because the t distribution takes into account the number of samples used such that it for a low number of samples have relatively many scores in the tails, i.e. one have to extend farther for the mean to contain a given percentage of the area. While the t distribution, as the number of sample increase, comes closer to the normal distribution. This means that the t distribution gives a more pessimistic result compared to the normal distribution, which is a desirable property since both M and s_m are estimations of the actual population and could potentially be outliers, thus the t distribution takes outliers into account [18, sec. VIII.E.3].

For our purposes we can look Z_{cl} up in a table such as the one found in [18, sec. VIII.E.3]. If we for example have 11 samples and want a confidence level of 95% then $Z_{cl} = 2.228$.

C Our Benchmark Suite

In the following, we will describe the benchmarks which we have chosen for this project. Since the benchmarks will be used as test cases for APL, we will start with some very simple benchmarks, which we should be able to run in the first couple of iterations of our development. Afterwards, we will move on to more complex benchmarks. We will base most of our benchmarks on the code examples found in the CUDA SDK, since this allows us to better compare our solution against the handwritten CUDA C samples. The sequential for-loop, TPL and APL have been implemented in C#. This information is used in Section 4.1.1and Section 4.1.2.

C.1 Overhead

This is the simplest benchmark in the benchmark suite. It runs a number of iterations on a loop with an empty body. This benchmark is interesting since there might be a significant amount of overhead involved with kernel invocation on the GPU, or moving different iterations of a loop to different threads. The APL and TPL implementation of this benchmark is shown in Figure C.1.

```
Parallel.For(0, size, delegate(int i)
{
});
```

Figure C.1: Overhead Benchmark in APL and TPL

C.2 Vector Addition

This benchmark adds corresponding elements from two vectors together to produce a third vector. This is a benchmark which is very memory dependent, since it requires at least three memory accesses for each addition. Furthermore, the Peripheral Component Interconnect Express (PCIe) bus could become a bottleneck for the GPU since a lot of data have to be transfered to and from the

GPU, with very few computations being performed on the GPU per data element. Figure C.2 shows the APL and TPL implementation of this benchmark.

```
Parallel.For(0, size, delegate(int i)
{
  result[i] = v1[i] + v2[i];
});
```

Figure C.2: Vector Addition Benchmark in APL and TPL

C.3 Matrix Multiplication

This benchmark somewhat more complex than the previous benchmarks, since it makes use of control flow inside the outer loop. The implementations should use a $O(n^3)$ algorithm, like the one used in the CUDA SDK. This means that there is also a lot more compute work to be done on the GPU, without requiring data to be transferred over the PCIe bus. This should result in less of a bottleneck than in the vector addition benchmark. The TPL implementations of this benchmark is given in Figure C.3.

Note that the TPL implementation will also work in APL, however, due to this implementation only spawns *size* threads, were *size* is the size of the dimensions, i.e. 768. We have rewritten the matrix multiplication as seen on Figure C.4. Here, n^2 threads are spawned and an accumulator is used which reduces the number of global memory accesses.

```
Parallel.For(0, size, delegate(int i)
{
  for (int j = 0; j < size; j++)
  {
    result[i, j] = 0;
    for (int k = 0; k < size; k++)
    {
      result[i, j] += m1[i, k] * m2[k, j];
    }
  }
});
```

Figure C.3: Matrix Multiplication Benchmark in TPL

```
Parallel.For(0, size*size, delegate(int index)
{
    int i = index % size;
    int j = index / size;
    float acc = 0.0f;
    for (int k = 0; k < size; k++)
    {
        acc = acc + (m1[i, k] * m2[k, j]);
    }
    result[i, j] = acc;
});
```

Figure C.4: Matrix Multiplication Benchmark optimized for APL

C.4 Black Scholes

Black Scholes is a method for calculation fair call and put prices for a set of European options. It makes use of more complex arithmetic operations then the previous benchmarks, like the `log` and `exp` operations. It is also often implemented using function calls like in the implementation found in the CUDA SDK. The Black-Scholes method has a complexity of $O(n)$, and as such has low arithmetic complexity compared to Matrix Multiplication. There is however a large amount of calculations that needs to be performed for each input element, and we therefore expect to see better performance with APL then TPL.

The APL and TPL implementations of this benchmark is given in Figure C.5.

```
...
...
Parallel.For(0, size, delegate(int j)
{
  float d1 = (((float)(Math.Log(StockPrice[j] / OptionStrike[j])) +
       (riskfree + volatility * volatility / 2) * OptionYears[j]) /
       (volatility *((float)Math.Sqrt(OptionYears[j]))));
  float d2 = d1 - volatility *((float) Math.Sqrt(OptionYears[j]));
  CallResult[j] = StockPrice[j] * CND(d1) -OptionStrike[j] *((float
      ) Math.Exp(-riskfree*OptionYears[j])) * CND(d2);
  PutResult[j] = OptionStrike[j] * ((float) Math.Exp(-riskfree*
      OptionYears[j])) * (CND(-d2)) - StockPrice[j] * (CND(-d1));
});
...
...
private static float CND(float d)
{
  float A1 = 0.31938153f;
  float A2 = -0.356563782f;
  float A3 = 1.781477937f;
  float A4 = -1.821255978f;
  float A5 = 1.330274429f;
  float RSQRT2PI = 0.39894228040143267793994605993438f;
  float K = 1.0f / (1.0f + 0.2316419f * Math.Abs(d));
  float cnd = RSQRT2PI * (float)Math.Exp(-0.5f * d * d) * (K * (A1
      + K * (A2 + K * (A3 + K * (A4 + K * A5)))));
  if (d > 0)
    cnd = 1.0f - cnd;
  return cnd;
}
```

Figure C.5: Black Scholes Benchmark in APL and TPL

D Benchmark Runner

To run our benchmark suite automatically, we have created a benchmark runner. The implementation of this is described in the following, first we will introduce the different classes that make up the benchmark runner and then give a more detailed description of the sequence of operations performed by the benchmark runner. This information is used in Table 4.1.

D.1 Class Diagram

Figure D.1 depicts the class diagram of the benchmark runner which consists of three namespaces:

- `Benchmarks`
- `Benchmarks.Implementations`
- `Benchmarks.Utilities`

These namespaces and their contents are described in the following.

D.1.1 Benchmarks Namespace

The `Benchmarks` namespace is the primary namespace of the benchmark runner.

BenchmarkSuiteRunner

The `BenchmarkSuiteRunner` is the main entry point for running the benchmark suite. It is implemented as a PowerShell script which runs the benchmark suite and asserts whether or not all benchmark implementations produce the same results.

ConfidenceIntervalCalculator

The `ConfidenceIntervalCalculator` is an application that takes a file containing the measurements from one benchmark run and prints the computed confidence interval for the measurements.

IBenchmarkSuiteImplementation

The IBenchmarkSuiteImplementation defines a command line interface which all the platform specific, i.e. CUDA C, TPL, etc. implementations of the benchmark suite must follow. It defines the Count operation which prints the number of benchmark implementations for a given platform, the Startup operation runs a single iteration of a benchmark implementation and prints the time it took in millisecond(ms), the SteadyState operation runs a benchmark implementation until steady state has been detected for twenty consecutive iterations and prints the time it took for each of these iterations in ms. The CreateTestData operation prints the data produced by running a benchmark implementation, such that correctnesses can be asserted, and the Assert operation asserts that the results produced by running a benchmark implementation match the results in a given file.

D.1.2 Benchmarks.Implementations Namespace

The Benchmarks.Implementations namespace contains a number of applications which implement the IBenchmarkSuiteImplementation interface. Each of these applications contains all the benchmark implementations for a given platform, e.g. CUDA C or TPL. Each benchmark implementation implements the IBenchmark interface.

D.1.3 Benchmarks.Utilities Namespace

The Benchmarks.Utilities namespace contains a number of classes that are shared between the different applications in the Benchmarks namespace, such as the confidence interval calculator and each benchmark suite implementation which are all executables.

CommandLineParser

The ParseArguments operation on CommandLineParser takes an array of command line arguments and an array of benchmark implementations, all implementing the IBenchmark interface, in order to perform the operations defined by the IBenchmarkSuiteImplementation interface.

Calculations

The Calculations class contains a number of operations for doing calculations used by other parts of the benchmark runner. The two most important of these are the ConfidenceInterval and Variance, which respectively calculates the confidence interval as defined in Section B.3.1 and coefficient of variance as defined in Section B.2.6, for a given number of samples.

BenchmarkRunner

The BenchmarkRunner contains the operations Startup and SteadyState which measures the performance of a given benchmark implementation, i.e. an implementations of IBenchmark which is given as a parameter to the operation, at startup and steady state respectively.

IBenchmark

`IBenchmark` is an interface which all benchmark implementations must implement. The operations this interface defines are: `CreateResults`, prints the results of running the benchmark, `Run`, runs the benchmark and returns the time used for execution on the GPU, and `Assert`, asserts whether or not the benchmark implementation produces the same results as those found in a given file. The `Run` method return zero on both TPL and sequential-for loop, since neither of these utilizes the GPU.

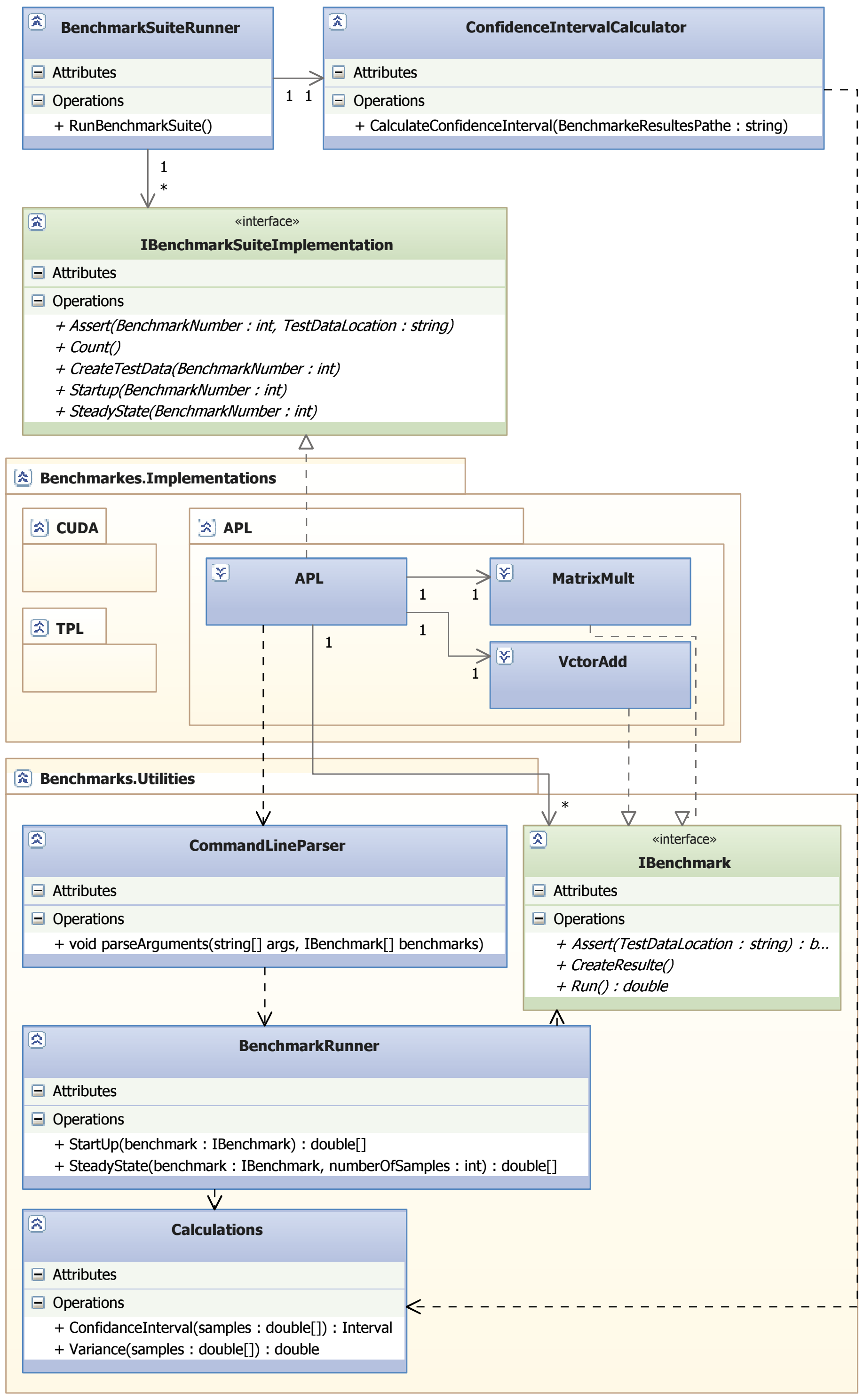

BenchmarkSuiteRunner
Attributes
Operations
+ RunBenchmarkSuite()
ConfidenceIntervalCalculator
Attributes
Operations
+ CalculateConfidenceInterval(BenchmarkeResultesPathe : string)
1
1
1
*
«interface»
IBenchmarkSuiteImplementation
Attributes
Operations
+ Assert(BenchmarkNumber : int, TestDataLocation : string)
+ Count()
+ CreateTestData(BenchmarkNumber : int)
+ Startup(BenchmarkNumber : int)
+ SteadyState(BenchmarkNumber : int)
Benchmarkes.Implementations
CUDA
TPL
APL
APL
MatrixMult
VctorAdd
1
1
1
1
1
Benchmarks.Utilities
*
CommandLineParser
Attributes
Operations
+ void parseArguments(string[] args, IBenchmark[] benchmarks)
«interface»
IBenchmark
Attributes
Operations
+ Assert(TestDataLocation : string) : b...
+ CreateResulte()
+ Run() : double
BenchmarkRunner
Attributes
Operations
+ StartUp(benchmark : IBenchmark) : double[]
+ SteadyState(benchmark : IBenchmark, numberOfSamples : int) : double[]
Calculations
Attributes
Operations
+ ConfidanceInterval(samples : double[]) : Interval
+ Variance(samples : double[]) : double

Figure D.1: Class diagram of the Benchmark Runner

D.2 Sequence Diagram

We have split the description of the sequence of operations performed by the benchmark runner in to two parts. First, we will describe the sequence of operations performed by `BenchmarkSuiteRunner` and then describe the sequence of operations performed by the APL implementation of the `IBenchmarkSuiteImplementation` interface.

D.2.1 BenchmarkSuiteRunner

The sequence diagram of `BenchmarkSuiteRunner` is divided in two figures: Figure D.2 and Figure D.3. As shown, the `BenchmarkSuiteRunner` iterates over all applications that implement the `IBenchmarkSuiteImplementation`.

For each of these iterations, the number of benchmarks implementations in a given `IBenchmarkSuiteImplementation` is determined by calling `Count`. Each of these benchmark implementations is run in steady state `NUMBEROFSTEADYSTATERUNS` times by repeatedly calling `SteadyState`. The result of calling `SteadyState` is piped to a file such that they can processed later.

Once the steady state runs of a benchmark implementation has been completed, the `ConfidenceIntervalCalculator` application is used to calculate the confidence interval of the measurements in the file, to which the results of calling `SteadyState` was piped.

Next, we call `Startup` to ensure that disk access does not affect our benchmarks results when measuring startup performance as described in Section 2.4.1. After this single call, we call `Startup` an additional `NUMBEROFSTARTUPRUNS` times where we pipe the output of each run to a file. Once all the `NUMBEROFSTARTUPRUNS` runs are completed, the `ConfidenceIntervalCalculator` is used to calculate the confidence interval for the startup runs.

When the confidence interval has been calculated the `CreateTestData` operation is called, and the results are piped to a file so the correctness of the results produced by the benchmarks implementations can be asserted later.

Once all the `IBenchmarkSuiteImplementation` implementations have been processed, another loop is entered, which iterates over all `IBenchmarkSuiteImplementation` implementations. Inside this loop, all the benchmark implementations in each of the `IBenchmarkSuiteImplementation` is iterated over, and for each of these the correctness of the results are asserted by calling the `Assert` operation once for each of the files containing the results produced by the other implementations of the benchmark implementations.

D.2.2 IBenchmarkSuiteImplementation

There are multiple applications which implement the `IBenchmarkSuiteImplementation` interface, but all of these perform the same basic sequence of operations, therefore we will only present the implementation which targets the APL here. The sequence diagram for the APL implementation is shown in Figure D.4 and Figure D.5.

The first thing the `main` function does is to instantiate the different benchmark implementations. These are then passed as arguments to `ParseArguments` in the form of an array of `IBenchmark` implementations, along with the command line arguments that were given to the application. `ParseArguments` checks the first command line argument to determine which of the operations, defined in the `IBenchmarkSuiteImplementation` interface, should be performed.

If the first argument is "count", the length of the array of `IBenchmark` implementations is written to the console.

If the first argument is "steadystate", the `SteadyState` operation on `BenchmarkRunner` is called with the `IBenchmark` indicated by the second command line argument. While the coefficient of variance is greater than `DESIREDCOEFFICENTOFVARIANCE`, `SteadyState` repeatedly calls `Run` on the `IBenchmark` and measures the time this took. If more than twenty iterations of the loop have been completed, the `Variance` operation on `Calculations` is called to calculate the coefficient of variance over the last twenty measurements. When the while loop exits, the last twenty measurements are returned and printed to the console.

If the first argument is "startup", the `Startup` operation on `BenchmarkRunner` is called with the `IBenchmark`, indicated by the second command line argument. This will measures the time it takes to execute a single call to `Run` and returns this measurement which is then printed to the console.

If the first argument is "createdata", the `CreateResults` operation is called on the `IBenchmark` indicated by the second command line argument. `CreateResults` first runs the task in the given benchmark and then prints the results of the tasks to the console.

If the first argument is "assert", the `Assert` operation is called on the `IBenchmark`, indicated by the second command line argument, with the file path given in the third command line argument as argument. `Assert` first runs the task of the benchmark and then compares the results of running this task with the results in the file given as argument to the operation. This operation prints a boolean value to the console, where the value indicates if the produced benchmark results is equal to the results at the file path.

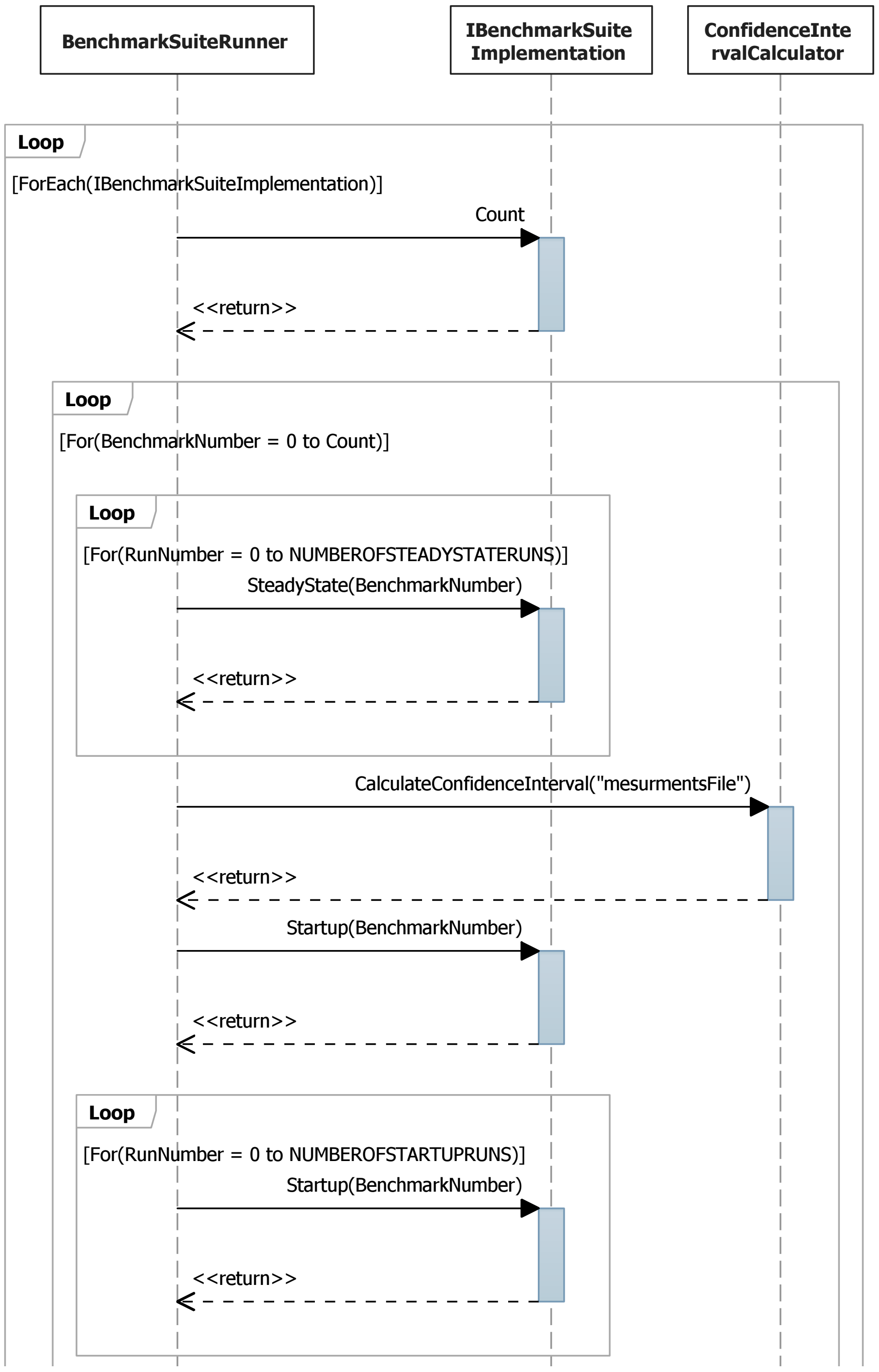

BenchmarkSuiteRunner
IBenchmarkSuite Implementation
ConfidenceIntervalCalculator
Loop
[ForEach(IBenchmarkSuiteImplementation)]
Count
<<return>>
Loop
[For(BenchmarkNumber = 0 to Count)]
Loop
[For(RunNumber = 0 to NUMBEROFSTEADYSTATERUNS)]
SteadyState(BenchmarkNumber)
<<return>>
CalculateConfidenceInterval("mesurmentsFile")
<<return>>
Startup(BenchmarkNumber)
<<return>>
Loop
[For(RunNumber = 0 to NUMBEROFSTARTUPRUNS)]
Startup(BenchmarkNumber)
<<return>>

Figure D.2: Part 1 of the sequence diagram of the Benchmark Suite Runner

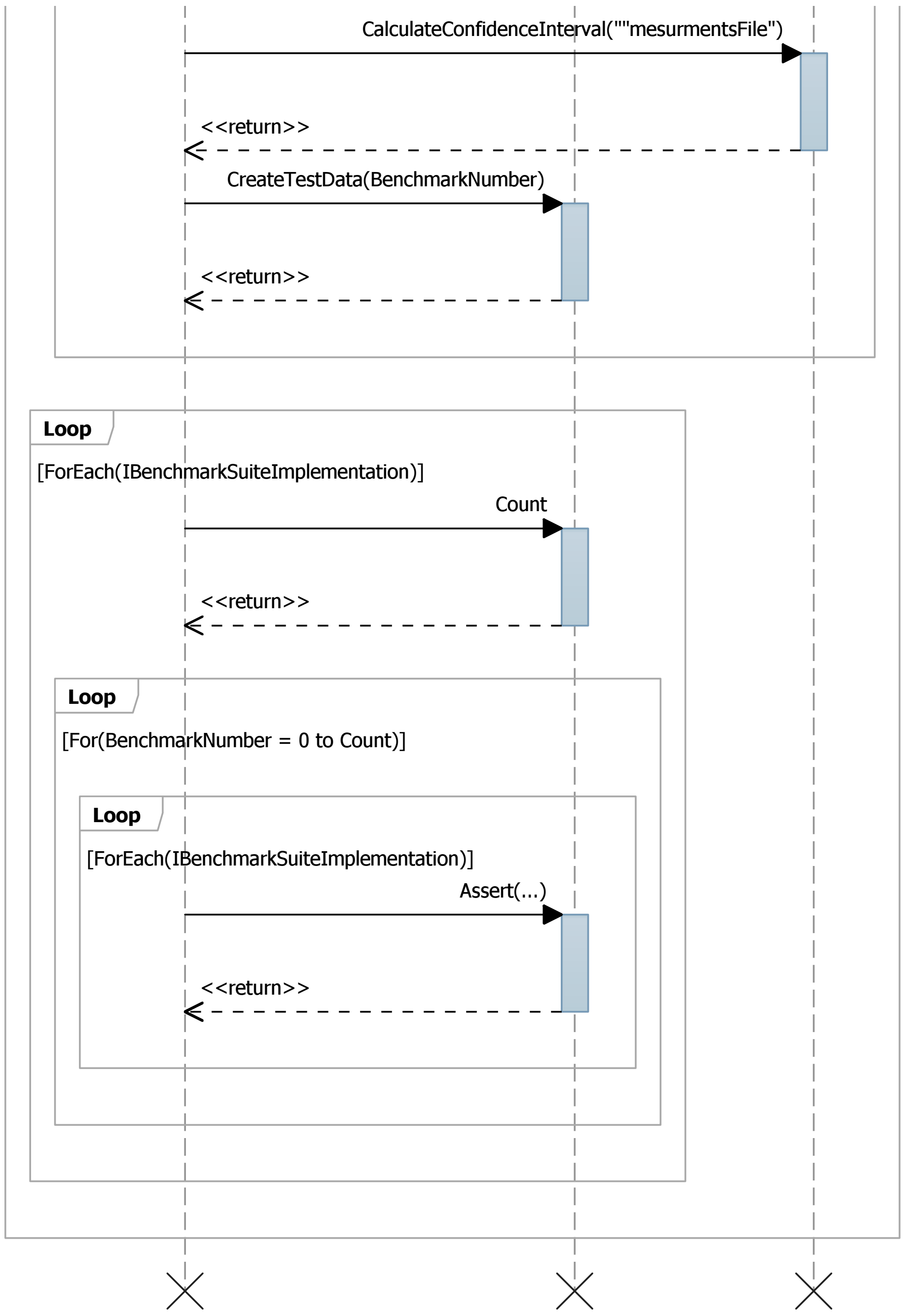

CalculateConfidenceInterval(""mesurmentsFile")
<<return>>
CreateTestData(BenchmarkNumber)
<<return>>
Loop
[ForEach(IBenchmarkSuiteImplementation)]
Count
<<return>>
Loop
[For(BenchmarkNumber = 0 to Count)]
Loop
[ForEach(IBenchmarkSuiteImplementation)]
Assert(...)
<<return>>

Figure D.3: Part 2 of the sequence diagram of the Benchmark Suite Runner

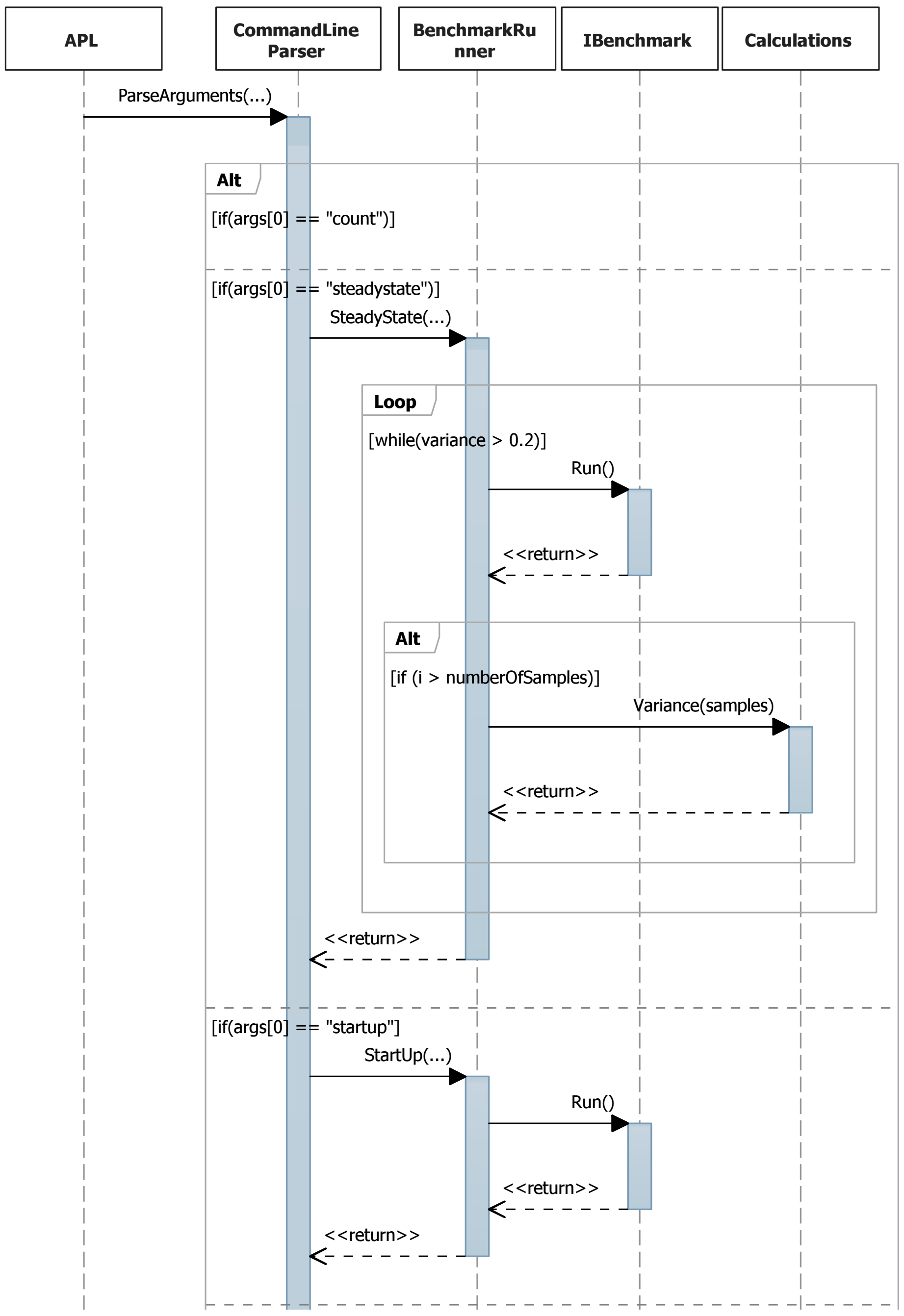


Figure D.4: Part 1 of the sequence diagram of the APL Benchmark Suite implementation

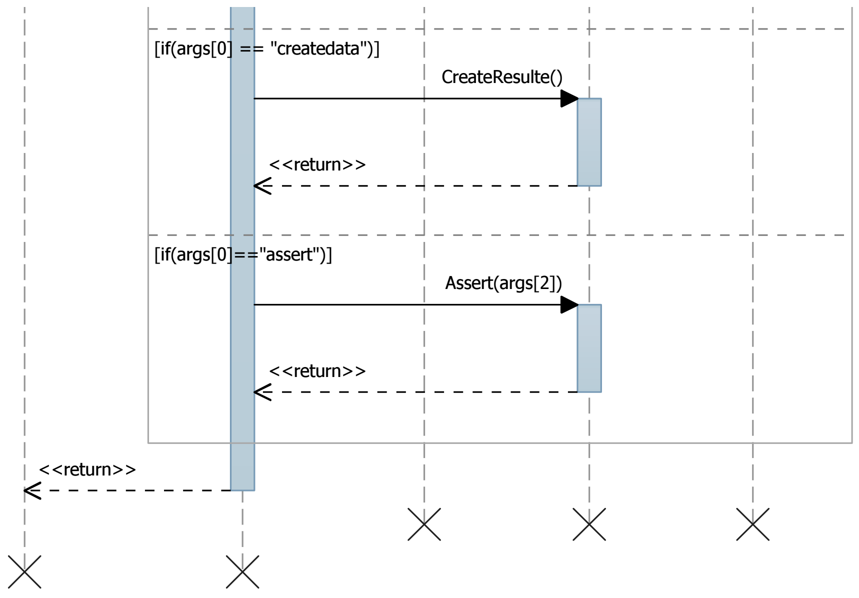


Figure D.5: Part 2 of the sequence diagram of the APL Benchmark Suite implementation

E CUDA Compute Capability

There exists different CUDA compute capabilities which define what features a given GPU supports. Two major versions of CUDA compute capabilities currently exists, compute capability 1.x and 2.x. These corresponds to the Tesla architecture and the Fermi architecture respectively. Some of the information presented in this chapter is used in Section 3.3.3 and Section 4.6.1.

The Tesla architecture should not be confused with the Tesla branded GPUs, such as the Tesla C870 and Tesla C2070, since these implement either the Tesla or Fermi architecture. [41, p. 1]

The Fermi architecture was introduced in 2010 with the GF100 chip, and it replaces the Tesla architecture, which was released in 2006 with the G80 chip. The Fermi architecture is used in all compute capability 2.x GPUs and features a number of changes compared to the Tesla based GPUs with compute capability 1.x. These changes should be taken into consideration to ensure high performance.

Programs which use compute capability 2.x or 1.x must exhibit many of the same characteristics to achieve high performance. In the following we will give a short overview of the characteristics all applications must share regardless of the target compute capability, followed by a more detailed look at what should be done to achieve high performance with compute capability 2.x.

E.1 Best Practices

There are a number of best practices that should be followed when developing applications using CUDA enabled GPUs. We will give a quick overview of the most important of these as defined by [48], for a more complete description see [16] or [47].

One of the most important things to consider in order to achieve good performance when developing applications which use CUDA enabled GPUs, is is to parallelize the code. This is because GPUs need to execute hundreds or thousands of concurrent threads to achieve good performance.

Communication between the host and the device should be minimized to avoid the PCIe bus forming a bottleneck since the PCIe bus, which usually connects the host and device, has a comparatively low bandwidth for transferring data between device- and host memory.

GPUs relies on latency hiding to eliminate time spend waiting for resources, i.e. executing instructions from another thread when waiting for high latency operations to complete, which requires that each SM can execute a large number of concurrent threads to be effective. However, the number of concurrent threads a SM can execute depends on several factors, such as the thread layout. The chosen launch configuration of kernels should therefor try to achieve the highest possible utilization of each SM.

Data in global memory should be accessed in a pattern that allows multiple accesses to be coalesced in a single memory transaction. This is because un-coalesced memory accesses waste a lot of bandwidth, since it transfers data that is never used. This decreases the bandwidth available to the application.

Since shared memory has a much higher bandwidth and lower latency than global memory, accesses to global memory should be replaced with accesses to shared memory whenever possible.

Threads in a warp execute in lock step, i.e. all active threads in a warp execute the same instruction at the same time. Thus, if different threads in a warp follow different branches, these branches will be executed sequentially leading to a decrease in performance. Different execution paths in waprs should therefore be avoid.

E.2 Compute Capability 1.x vs Compute Capability 2.x

The following is based on [41] and [48]. There have been a number of changes between compute capability 2.x and compute capability 1.x which should be taken into account when developing for compute capability 2.x, especially if one has previously developed for compute capability 1.x.

Compute capability 2.x allows threads of different kernels to execute concurrently on the GPU, thus giving more flexibility and performance since different kernels can potentially utilize more SMs on the GPU. With compute capability 1.x, utilization of all SMs could only be achieved by running a kernel that used enough thread blocks to fill all SMs. Thus, compute capability 2.x allows a limited form of task parallelism since multiple kernels can run concurrently, compared to compute capability 1.x which only allowed one kernel at a time.

With compute capability 2.x a L1 cache has been introduced on each SM which resides in the shared memory already found on the SMs. This L1 cache is used to cache accesses to global and local memory. The programmer can choose how large a portion of the shared memory is used for cache and how much is used for ordinary shared memory. The portion of the shared memory can have a large impact on performance and should therefore be considered carefully. The introduction of the L1 cache means that texture memory is less useful then previously where using texture memory could lead to significantly improved performance since it was cached. Using texture memory with compute capability 2.x can even lead to a decrease in performance, since the texture cache has a higher latency than shared memory. Texture memory is still useful in some cases since it can provide other benefits than caching, such as address calculations or texture filtering.

With compute capability 2.x, the number of registers per SM has been increased from 8192 to 32768. This increase allows more threads to run concurrently, thereby increasing the latency hiding ability of the device and increasing performance.

With compute capability 2.x, global memory access is processed a warp at a time, while it was processed a half warp at a time with compute capability 1.x. Therefore changing kernel launch configurations there assume a per half warp access pattern might lead to a performance increase if they are changed so that the x dimension of thread blocks have a size that is a multiple of the warp size as each warp could addresses a single cache line.

With compute capability 1.x, shared memory was split in 16 banks and access was processed a whole warp at a time. With compute capability 2.x there are 32 banks, and access is processed a whole warp at a time. This means that there can now be bank conflicts between threads in the first and the last half of a warp, and memory might need to be padded differently to avoid this. However compute capability 2.x has better support for broadcasting a read to multiple threads on the warp and is better at avoiding bank conflicts on non 32-bit word accesses.

In addition to constant memory which is managed by the programmer, compute capability 2.x features a `LDU` instruction that can be used by the compiler to load any variables as if it was stored in constant memory, as long as the variable is not a pointer to global memory, is read-only and not dependent on the thread id, thus gaining the advantages of constant memory i.e. caching and broadcasting of a single memory read to multiple threads.

Compute capability 2.x natively supports 32-bit integer multiplication while compute capability 1.x only supported 24-bit integer multiplication. 32-bit integer multiplication was implemented in compute capability 1.x using multiple instructions. The _[u]mul24 instruction could be used to improve performance when only 24-bit integers were required. Compute capability 2.x however do not natively support 24-bit integer multiplications, and therefore, the use of _[u]mul24 is implemented using multiple instructions, leading to lower performance.

Compute capability 2.x supports double precision floating points arithmetics, where compute capability 1.0-1.2 do not. Also, the double precision performance is much better with compute capability 2.x. Compute capability 2.x provides a factor 2/5 of the single precision performance when performing double precision computations, while the double precision performance of compute capability 1.3 were much lower, a factor 1/12.

The IEEE754-208 compliance has been improved with compute capability 2.x compared to compute capability 1.x. In particular, addition and multiplication were often combined it to a single `FMAD` instruction with compute capability 1.x in a none standard compliment way. With compute capability 2.x, they are instead combined into a `FFMA` instruction that is standard compliant. In addition to this, there are a number of other changes that can cause compute capability 2.x and compute capability 1.x to produce different results. By default, the *nvcc* compiler produces standard compliant code but can be configured to produce results which are closer to that of compute capability 1.x. This can be achieved by using the following flags: `-ftz=true` which flushes denormalized numbers to zero, `-prec-div=false` which gives less precise division and `-prec-sqrt=false` which gives less precise square root. Using these flags also tend to give higher performance.

Compute capability 2.x allows the use of C++ classes in kernels, recursion is supported on device functions, and it is possible to dynamically allocate part of a preallocated heap of memory to threads in a kernel.

An optimization sometimes used with compute capability 1.x is to take advantage of the fact that all treads in a warp execute in lock step, and explicit synchronization can sometimes be omitted without affecting the correctness of the program. With compute capability 2.x, accesses to a memory location can sometimes be cached in registers. Therefore, a change made to a memory location may not be reflected in another thread, since that thread would read from its register instead of global memory. This means that all memory that is used to communicate between threads, without using explicit synchronization, should be declared as volatile, as this ensures that it is not cached in registers. Note that data should also be declared volatile with compute capability 1.x but it had no affect on the execution of a program if it was not, a bit like a C program where 32 bit integers is used as pointer will work an a 32-bit CPU but fail on a 64-bit CPU.

F Summary

During our 9th semester project, we saw that computations could be moved from the Central Processing Unit (CPU) to the Graphics Processing Unit (GPU), using the techniques of General-Purpose computations on Graphics Processing Units (GPGPU). We also saw that GPGPU has been regarded as a "black art", and few posses the skills needed to implement GPGPU powered applications. Though programming languages such as Compute Unified Device Architecture (CUDA) C and Open Computing Language (OpenCL) C, and their compute models, provides some abstraction, we found that these abstractions are still relatively low-level. Looking beyond the 9th semester project, we have seen that the research community is working towards higher-level GPGPU abstractions.

With this in mind, we decided to increase the abstraction level of GPGPU programming from .NET languages, by providing the same abstractions as the Task Parallel Library (TPL). To this end, we introduced the Accelerated Parallel Library (APL) which provides the same abstractions as TPL while accelerating the computations by using the GPU. This choice of abstraction allows existing TPL powered applications to take advantage of the GPU, by changing only a few lines of code.

Prior to implementing APL, we found that in order to run .NET programs on the GPU we had to translate host code to device code. This meant that we had to choose a source language and a target language. We found that the .NET framework is a concrete implementation of the Common Language Infrastructure (CLI), and that the Common Language Runtime (CLR) is a concrete implementation of the Virtual Execution System (VES), which executes Common Intermediate Language (CIL) code. We also found that all high-level CLI languages are translated to CIL prior to execution on the VES. CIL was thereafter chosen as the source language, since this choice allowed APL to be used in multiple high-level languages.

Afterwards, we looked at how the official CUDA C compiler called *nvcc*, which compiles CUDA C host and device code, and found that *nvcc* producec Parallel Thread Execution (PTX) code which is embedded into the GPGPU powered application. We also found that the CUDA Driver Application Programming Interface (API) can load PTX kernels and Just-In-Time (JIT) compile them to the device present in the system. We therefore decided to target the PTX language.

With the source language being CIL and the target language being PTX, we looked at ways of translating between these two languages. We found that CIL is a stack-oriented language, and that PTX is a register-oriented language. To overcome this we chose to use Static Single Assignment (SSA), a technique which translates each push to the stack to an assignment to a new variable, and each pop from the stack to a read from the corresponding variable. APL uses reflection to access the low-level CIL code at runtime, thus we were able to JIT compile the code to be run on the GPU.

To drive the development of APL, we created four benchmarks: Overhead, Vector Addition, Matrix Multiplication and Black Scholes. Each added a new set of requirements to APL, e.g. the Vector Addition benchmark required support for single dimension arrays and arithmetic addition, while the Matrix Multiplication benchmark required rectangular arrays, arithmetic multiplication and branching.

We decided to split the architecture of APL into four components: the `Parser`, the `Optimizer`, the `Emitter` and the `Invoker`. The `Parser` was in charge of parsing the CIL code to produce an Intermediate Representation (IR). The `Optimizer` was in charge of performing optimizations on the IR. The `Emitter` was in charge of producing PTX instructions from the IR. And the `Invoker` was in charge of initializing the device, allocating device memory, marshalling data to and from the device, and invoking the kernel.

Furthermore, five optimizations were implemented. PTX Caching which was used to cache compiled kernels. More L1 Cache which increased the portion of shared memory used for L1 cache, thus more data could be cached on chip. CUDA Context Caching which avoided creating a new CUDA context for each call to APL, since context creation proved to be a time-consuming operation. Copy Omit which reduced redundant copies to and from the device. Lastly, Fused Multiply Add which, replaces a `mul` instruction followed by an `add` instruction with a single Fused Multiply Add (`mad`) instruction.

To investigate the performance of APL each benchmark was implemented using: APL, TPL, sequential for-loops and CUDA C. The results of running the benchmarks showed that all optimizations except PTX caching and Fused Multiply Add provided a measurable performance increase. The results also showed that APL beat CUDA C in two out of four benchmarks in steady state, with a speedup of 1.03x in the Vector Addition benchmark, and a speedup of 1.02x in the Black Scholes benchmark. APL beat TPL in all benchmarks, except for the Vector Addition benchmark, where TPL won due to the low arithmetic intensity of the benchmark and large overhead incurred by APL to transfer data to and from the device. We saw that the APL implementation of Black Scholes, Matrix Multiplication and Overhead outperformed TPL with speedups of 2.5x, 2.58x and 82.5x respectively. With regards to scaling, we saw that the APL benchmark implementations scaled just as well or better than the other benchmark implementations, except for the CUDA C Matrix Multiplication benchmark implementation, which was highly optimized to take advantage of the shared memory on the device.

Based upon the results, we concluded that APL did indeed provide the abstraction of TPL, and that it was a useful library for GPGPU application development. We did however also conclude, that further work was needed prior to release, such as supporting more opcodes, structures and classes.